Sistemas Informáticos
Curso 2002-03

Interfaz gráfica de un asistente pedagógico

Jorge Díaz Estebaranz
Javier Fernández Cruz
Alberto Ribao Valverde

Dirigido por:
Luis Hernández Yáñez
Dpto. de Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

Índice de contenidos

Resumen del proyecto

Español:

El objetivo de este proyecto es la creación de un interfaz gráfico tridimensional para un asistente pedagógico, el cual trata de enseñar al usuario una metáfora de la maquina virtual de java. El proyecto está implementado en C++ y hace uso del motor gráfico WildMagic. En el proyecto se implementan algoritmos gráficos y de inteligencia artificial de diversa índole, tales como búsqueda de caminos en mapas y grafos, detección y control de colisiones, carga de personajes, de mundos tridimensionales y control de la cámara.

English:

The aim of this project is the creation of a graphic three-dimensional interface for a pedagogical assistant, who tries to teach to the user a java virtual machine metaphor. The project is implemented in C++ and uses the graphic engine WildMagic. Graphic, artificial intelligence and others algorithms are implementated in this project, just as problems of pathfinding in maps and graphs, collision detection and control, characters and three-dimensional worlds loading and camera control.

Capitulo 1
Introducción

El ciclo y la labor de aprendizaje de una persona abarcan gran parte de su infancia y adolescencia, e incluso en el caso de profesores y estudiantes universitarios gran parte de su vida madura. La formación de los futuros trabajadores de vital importancia para el destino de una sociedad, lo que representa una gran responsabilidad para los profesores.

La productividad de la educación siempre es un factor importante para la sociedad debido al coste económico y temporal asociado a la enseñanza. Mientras reciben este proceso de aprendizaje, las personas implicadas no suelen realizar ningún trabajo que sirva de ayuda a la sociedad. Para que esa inversión merezca la pena, a la larga esa persona tendrá que *producir* para devolver de algún modo lo que ha recibido como un préstamo anteriormente. Para que este modo de proceder funcione correctamente, la educación debe conseguir un alto porcentaje de éxito en su tarea de formar, razón que hace que el fracaso escolar sea considerado negativo, y un modo de medir el éxito de un sistema educativo.

Aparte de la enseñanza básica la sociedad necesita de personas cualificadas que puedan hacer tareas especializadas. La preparación de estos profesionales es también muy cara, y a menudo implica ciertos riesgos. Por ejemplo, el adiestramiento de pilotos pone en peligro material caro que podría resultar dañado. O, incluso aún peor, la formación de cirujanos supone un cierto riesgo para la salud de los enfermos.

La tecnología ha hecho avanzar rápidamente las capacidades de los ordenadores, que se han convertido en electrodomésticos disponibles en la mayoría de los hogares, muchas veces infrautilizados. Con el incremento de su poder de cálculo se han abierto nuevas posibilidades, entre las que se encuentra la capacidad de mostrar gráficos en tiempo real con una aceptable calidad. Además, la aparición de nuevos soportes de almacenamiento de información han disparado la cantidad de datos que pueden guardarse hasta límites insospechados hace sólo unos años. Gracias a ambos avances, los programas

interactivos multimedia han llegado a ser una realidad, sustituyendo en parte a las enciclopedias que tradicionalmente han ocupado una gran cantidad de espacio en las estanterías de todos los hogares.

Sin embargo, estos programas no han sido los primeros sistemas que han abordado el problema de la enseñanza por ordenador. Los simuladores informáticos han tenido gran repercusión en el modo de educar a ciertos profesionales, como pilotos de avión, al reducir costes y riesgos. Por su parte los sistemas denominados CAI (Computer Assisted Instruction) y su evolución, los ITS (Intelligent Tutoring System), se han utilizado en un intento de sustituir completamente a los profesores en ciertas áreas, habitualmente basadas en conocimiento procedural. De ese modo un ordenador puede ser capaz de enseñar a los alumnos el modo en el que llevar a cabo algunas tareas, mientras éstos las ponen en práctica en simuladores o entornos de aprendizaje adaptados.

La última tendencia en este tipo de enseñanza lo representan los agentes pedagógicos, que hacen uso de, al menos, dos prometedores campos de la informática: la realidad virtual, y los agentes.

El campo de la realidad virtual se encarga de la creación de entornos tridimensionales sintéticos generados por ordenador. El usuario se ve inmerso en uno de tales entornos, pudiendo desplazarse por él, e interactuar con los diferentes elementos que lo forman. La realidad virtual tiene muchos usos. Permite recrear, por ejemplo, lugares destruidos, o crear entornos artificiales donde la gente puede reunirse por medio de teleconferencias o donde se les permite practicar ciertas habilidades. Éste último caso convierte a estos entornos en entornos de aprendizaje, pues el usuario los utiliza para instruirse en un área determinada. He aquí donde se enmarca nuestro trabajo.

Por otro lado los agentes son sistemas software o hardware que habitan en un entorno con el que interactúan con la intención de satisfacer sus objetivos. Para eso, el sistema debe tener un modo de conocer el estado del entorno, y cierta inteligencia para reaccionar ante él del modo correcto para conseguir su propósito.

De la mezcla de los entornos de aprendizaje y los agentes nacen los agentes pedagógicos. Cada uno de estos agentes habita en un entorno de aprendizaje y posee una representación dentro de él a menudo por medio de un avatar. El usuario del sistema puede así ver cómo el agente se desplaza e interactúa con dicho entorno. El objetivo de estos agentes pedagógicos es facilitar la tarea al alumno, proporcionándole consejos cuando éste pida ayuda o se equivoque, y sustituyéndole completamente para ejecutar por sí mismo la tarea cuando el alumno sea incapaz de llevarla a cabo. El desarrollo de este tipo de sistemas aglutina todos los problemas existentes en la creación de agentes y entornos virtuales. Para que el resultado sea satisfactorio, el agente debe mostrar un comportamiento coherente, tener una representación fiel, sincronizar sus movimientos y acciones con sus explicaciones, etc.

Se han implementado varios agentes pedagógicos como por ejemplo Herman the Bug, COSMO, WHIZLOW, o STEVE. Para la realización de este proyecto hemos de presentar un nuevo agente pedagógico, llamado JAVY, habitante de un entorno virtual

que representa, mediante una metáfora, la máquina virtual de Java. Los objetos de este entorno tienen una estrecha relación con la máquina virtual de tal forma que el usuario aprenderá la estructura de la JVM *(Java Virtual Machin*e) haciendo uso de ellos. La intención del sistema no se limita a enseñar la organización de la JVM y su "lenguaje ensamblador", sino que además pretende que el usuario aprenda cómo se lleva a cabo la compilación del código Java, así como que afiance sus conocimientos sobre las características de la programación orientada a objetos. Para eso el estudiante debe contar con conocimientos básicos de programación en Java.

La función del agente pedagógico dentro de todo esto es ayudar al usuario en aquellas partes donde encuentre problemas, proporcionando la base teórica que necesite y los consejos que le hagan falta en cada momento. Para eso JAVY debe poseer una cantidad importante de conocimiento respecto a la máquina virtual y al proceso de compilación de programas escritos en Java.

Por lo tanto el objetivo en el que se centrará este proyecto será dotar de un entorno virtual tridimensional al agente pedagógico Javy que permita la interacción del usuario para mejorar su aprendizaje y que sea lo mas amigable posible.

En primer lugar en el capitulo 2 hablaremos sobre el motor gráfico en el que nos hemos apoyado para llevar la tarea de realizar el entorno virtual donde se moverá el agente pedagógico (WildMagic) y en el Capitulo 3 entraremos en detalle en los algoritmos más importantes realizados, tanto de informática grafica como de inteligencia artificial, tales como búsqueda de caminos, control de colisiones, carga de escenarios y personajes etc.

Capitulo 2
Motor Gráfico

2.1 Introducción

Para llevar a cabo la tarea de realizar un entorno tridimensional para un agente pedagógico no hemos partido de cero, ya que nos apoyamos en un motor gráfico 3D.

Una vista clásica de la labor que realiza un motor gráfico es el renderizado o dibujado de polígonos. Ciertamente este es un componente necesario pero no es el único. Visto como una caja negra el moro se puede ver como un productor consumidor. Consume triángulos y produce como salida gráficos renderizados en el dispositivo de salida. Como consumidor puede ser alimentado con muchos más datos muy rápidamente en un corto espacio de tiempo o puede estar a la espera de que le lleguen datos para ser pintados.

Es necesario un proceso intermedio para el control de los datos de entrada al proceso encargado de pintar (renderer); el encargado de realizar esta labor es el gestor del grafo de la escena. La función principal del gestor del grafo de la escena es proveer triángulos al renderer pero la decisión de que triángulos se envían y cuales no recae sobre el gestor. Cuanto más realistas sean los objetos de la escena más complejo es el proceso de decidir que triángulos se envían al renderer para su dibujado.

Un motor gráfico también debe encargarse de las transformaciones geométricas de los objetos presentes en la escena así normalmente incluyen manejo de matrices de transformación, sistemas de coordenadas, quaternions, métodos de calculo de distancias etc. Y también de algoritmos generales tales como representación jerárquica de la escena, detección de colisiones, intersecciones entre objetos, representación de curvas y efectos especiales.

El motor gráfico elegido a sido WildMagic (http://www.wild-magic.com/) de David H. Eberly, motor freeware bastante flexible y potente aparte de adaptable y potable a otras plataformas (Win, Unix, MacOs, etc.)

A continuación detallaremos las características más destacadas que presenta este motor.

2.2 Características del motor

A continuación pasaremos a listar las características más reseñables del motor WildMagic:

- Métodos geométricos
  - Transformaciones
    - Escalado
    - Rotación
    - Traslación
    - Transformaciones homogéneas
  - Sistemas de coordenadas
  - Quaternions
    - Rotación
    - Interpolación
  - Ángulos de Euler
    - Factorización de matrices de rotación
    - Factorización de matrices de rotación en productos de dos términos
  - Objetos 3D estándar
    - Esferas
    - Cajas orientadas
    - Cápsulas
    - Grageas
    - Cilindros
    - Elipsoides
  - Métodos para calcular distancias
    - Punto a línea
    - Línea a línea
    - Punto a triángulo
    - Línea a triángulo
    - Punto a rectángulo
    - Línea a rectángulo
    - Triángulo a triángulo
    - Triángulo a rectángulo
    - Rectángulo a rectángulo
    - Punto a caja orientada
    - Punto a elipse
    - Punto a elipsoide
    - Punto a Circulo en 3D

- Representación jerárquica de la escena
    - Representación basada en árbol
        - Transformaciones locales
        - Transformaciones generales (mundo)
        - Volúmenes delimitadores
        - Estado de la renderización
        - Animación
    - Actualización del grafo de la escena
        - Unión de dos esferas
        - Unión de dos cajas orientadas
        - Unión de dos cápsulas
        - Unión de dos grageas
        - Unión de dos cilindros
        - Unión de dos elipsoides
        - Algoritmo para actualización del grafo de la escena
    - Renderizado del grafo de la escena
        - Algoritmos de eliminación de caras ocultas
        - Eliminación de caras ocultas mediante volúmenes delimitadores (bounding volumes)
- Selección de objetos 3D en su representación 2D
    - Algoritmos de intersección línea-objeto
    - Selección de múltiples objetos
- Detección de colisiones
    - Intersección de objetos dinámicos y líneas
    - Intersección de objetos dinámicos y planos
    - Intersección de objeto estático - objeto estático
    - Intersección de objeto dinámico - objeto dinámico
    - Procesamiento de la rotación y el movimiento de los objetos
    - Árboles de bounding box
    - Sistema simple de de detección de colisiones dinámico
- Generación de curvas
    - Curvas de bezier
    - Splines
    - Splines no paramétricos
    - Kochanek-Bartels Splines
- Generación de superficies
    - Parches de bezier rectangulares
    - Parches de bezier triangulares
    - Superficies de bezier cilíndricas
    - Parches de bezier rectangulares no paramétricos
    - Superficies cuadricas

- Animación de caracteres
    - Animación mediante keyframes
    - Cinemática inversa
- Generación de terrenos
- Ordenación espacial
    - Quadtrees y octrees
    - Portales
    - Partición binaria del espacio (BSP)
        - Construcción de árbol BSP
        - Eliminación de caras ocultas mediante BSP
        - Detección de colisiones mediante BSP
        - Selección 3D mediante BSP
- Efectos especiales
    - Lens flare
    - Enviroment mapping
    - Bump mapping
    - Proyección de luces
    - Proyección de sombras
    - Sistemas de partículas

Capitulo 3
Algoritmos

3.1 Búsqueda de caminos:

Añadir animaciones a los personajes que habitan en un entorno virtual supone una mejora sustancial sobre su representación, que los hace mucho más amigables al usuario que un simple modelo estático. Pero si el personaje es controlado automáticamente por el sistema (no es el avatar de un usuario) añadir animaciones no lo es todo. Idealmente, hay que conseguir que los personajes demuestren cierto grado de inteligencia. Y el primer nivel de *inteligencia* debe encargarse de que la interacción con su entorno sea coherente.

Posteriormente podremos añadirle capacidad de hablar, reconocer nuestros problemas, ayudarnos en nuestro trabajo, o cualquier cosa que consigamos; pero si, a pesar de todo eso, cuando el personaje tiene que desplazarse por el entorno parece dudar, rectifica su camino, o incluso no llega a él en absoluto, toda la credibilidad conseguida se desvanecerá.

La búsqueda de caminos *(pathfindin*g) viene a solucionar este problema. Su misión es construir el recorrido más adecuado para un personaje que se encuentra en una posición y desea llegar a otra, teniendo en cuenta la distancia recorrida, los obstáculos del entorno, el tipo de terreno atravesado, etc.

La solución más sencilla es que en cada momento el personaje decida en qué dirección ir de una forma aleatoria, pero desde luego no es muy eficaz. Otras soluciones seleccionan la dirección más prometedora en función de la posición actual y el destino, y sólo la modifican cuando se llega a un obstáculo. Estas técnicas se han copiado del área de la robótica, y tienen la ventaja de no necesitar conocer el mapa del entorno, pero en general no encuentran el camino óptimo.

Por lo tanto, habitualmente los algoritmos utilizados para solucionar este problema son adaptaciones de aquellos ampliamente estudiados por la Inteligencia Artificial sobre búsqueda de soluciones en espacios de estados.

El primer paso es, naturalmente, decidir cual será el espacio de estados. Una solución habitual es dividir el entorno en una cuadrícula. A no ser que el personaje pueda volar, esa división origina una matriz bidimensional. Cada celda se marca como ocupada o como libre. Cada estado representa la posición del personaje dentro de esa cuadrícula. El estado inicial será aquel en el que el personaje está en la posición de partida, y en el estado objetivo estará en la posición final.

Una vez definido el espacio de estados, podemos hacer uso de una gran variedad de algoritmos, como búsqueda en anchura, en anchura bidireccional, en profundidad (iterativa o no) y el algoritmo de Dijkstra. Es sabido que ninguno de ellos utiliza heurística, por lo que pierden mucho tiempo explorando nodos (estados) que son poco o nada prometedores.

Naturalmente la solución es utilizar el algoritmo A* al que se le proporciona una heurística estimando el coste desde el estado actual hasta el objetivo. Es sabido que este algoritmo tiene muy buenas características (si existe solución la encuentra, y es óptima si la heurística es admisible, y además no hay ningún otro algoritmo que examine menos estados con esa heurística).

El uso del algoritmo A* requiere (como algún otro) dar un coste al paso de un estado a otro. Podemos limitarnos a dar como coste la distancia recorrida, pero también se pueden hacer cosas más sofisticadas, como tener en cuenta el tipo de terreno que el personaje está recorriendo, para por ejemplo *disuadirle* (pero no *prohibirl*e) que pise un hipotético césped. Con esta misma técnica podríamos incluso hacer que el coste entre ir de un lugar A a otro B sea diferente que ir desde B a A para simular cambios de pendiente.

Ligeros retoques en la heurística utilizada permiten en ocasiones realizar un ajuste fino de la búsqueda. Por ejemplo, almacenando en el estado actual tanto la posición del personaje como la dirección que llevaba cuando llegó a ella, podemos hacer que la heurística considere a los pasos que requieren un cambio de dirección ligeramente más costosos que a aquellos que no la cambian. Esta mínima modificación hará que A* visite más estados, pero construirá caminos con menos cambios de dirección (que seguirán siendo óptimos en número de pasos).

Si el sistema es muy crítico en tiempo, se podrían utilizar heurísticas no admisibles. El camino encontrado quizá ya no sea óptimo, pero es posible que con esa modificación el algoritmo A* visite menos estados, con lo que utilizará menos tiempo para encontrar un camino.

Como se ha comentado, el algoritmo A* es óptimo para la heurística proporcionada. Es decir no hay ningún otro algoritmo que visite menos estados haciendo uso de esa heurística. Sin embargo, en búsqueda de caminos en tiempo real hay ocasiones en las que sigue siendo ineficiente.

Naturalmente, eso no es culpa del algoritmo, si no del modo de utilizarlo.

Por ejemplo, en un entorno de gran tamaño, la división en una cuadrícula puede original millones de celdas, que ocasionan unas listas de estados abiertos y cerrados durante el algoritmo inmensas, a parte de comprobar muchos estados inútiles. Un ejemplo extremo que lo demuestra es utilizar el algoritmo A* sobre un plano de Europa, si se desea ir desde Madrid a Atenas. Obviamente, el algoritmo recorrerá buena parte de la Península Ibérica y seguramente también de Italia antes de llegar.

Una solución utilizada en ocasiones es la búsqueda de caminos de forma jerárquica. La idea es disponer de un mapa más general, en el que se indique la conexión entre grandes territorios. En el ejemplo anterior, almacenaríamos, por ejemplo, un grafo con los países como nodos, y las aristas uniendo aquellos países que hagan frontera. Antes de ejecutar A* sobre el mapa completo, lo ejecutamos sobre el grafo general para conocer los países por los que deberíamos pasar. Luego ejecutamos A* sobre cada uno de esos países, desde la posición de entrada hasta la de salida en la frontera [1] (o hasta el destino, si estamos en el último país). En la situación anterior, A* recorrería España directamente hacia el norte, y no descendería por Italia.
Esta idea ahorra mucho espacio de búsqueda, pero puede suponer caminos mejorables al no elegir la mejor salida por la frontera en función del camino posterior dentro del siguiente país. En este sentido, posiblemente el algoritmo anterior atravesaría la frontera entre España y Francia por su parte central, en lugar de por el Este. El espacio de estados puede organizarse de otras formas en lugar de con una cuadrícula, por ejemplo utilizando quadtrees o grafos, en cuyo caso hay varias técnicas para colocar los nodos dentro del espacio. Utilizar grafos tiene la ventaja de requerir menos espacio (idealmente habrá muchos menos nodos, por lo que además será más rápido), y de manejar el espacio como un continuo. Sin embargo es más difícil añadir obstáculos dinámicos
El grafo puede precalcularse para ser usado directamente por el personaje, o puede ser construido por éste en tiempo de ejecución utilizando visión artificial ([MCF99]).
Si se está utilizando búsqueda de caminos jerárquica el camino podría construirse por pasos cada vez que se llega a una *frontera*. Eso podría suponer que el agente se parase al llegar a la frontera mientras se está ejecutando A*. Naturalmente eso se podría evitar calculando el siguiente camino un poco antes de que el agente llegue al cambio de zona.

Por otra parte los movimientos del agente pueden resultar un poco bruscos en los cambios de dirección. Si se está utilizando división del espacio en celdas se pueden utilizar splines de Catmull-Rom para redondear los cambios de dirección, o curvas Bézier si se colocan nuevos puntos de control manualmente.

Una forma diferente de calcular caminos es utilizar diagramas generalizados de Voronoi. Dado una serie de primitivas, denominadas espacios de Voronoi, un diagrama de Voronoi divide el espacio en regiones, cada una de ellas consistente en todos los puntos que tienen a una determinada primitiva más cerca que a cualquier otra (Figura 4.9).

Si se consideran a las primitivas como obstáculos, es fácil comprender que la separación de las regiones de Voronoi indican las posiciones más alejadas de ellos. Para la utilidad que le vamos a dar, no es necesario calcular con absoluta precisión las regiones de Voronoi: una aproximación discreta es suficiente. Es posible obtener tal aproximación de una forma muy eficiente haciendo uso de la aceleración disponible en las tarjetas gráficas actuales.

Utilizando la información proporcionada (tanto el diagrama como la distancia en cada punto a la primitiva más cercana) es posible construir caminos, incluso en entornos dinámicos con tal de recalcular el diagrama de Voronoi.

3.1.1 Algoritmo de búsqueda de caminos A*

3.1.1.1. Qué es:

El algoritmo A* (leído a-estrella) permite encontrar el camino entre dos puntos de un mapa (algoritmo de búsqueda de caminos). Lógicamente entre dos puntos de un mapa pueden existir muchos caminos diferente y todos validos, pero el algoritmo A* encontrará, si existe, el camino más corto (de menor coste) entre esos dos puntos y lo hará de una manera relativamente rápida comparado con otros algoritmos.

El algoritmo A* es un algoritmo directo, esto quiere decir que no hace una búsqueda ciega para encontrar un camino (como una rata en un laberinto), pero en vez de evaluar la mejor dirección a explorar , a veces da vuelta atrás para probar otros caminos alternativos

Hay muchas posibles implementaciones del algoritmo A* pero todas derivan del algoritmo básico que pasaremos a comentar a continuación.

3.1.1.2. Comó funciona:

Antes de entrar en detalle con las particularidades del algoritmo A* deberemos definir unos pocos términos:

Mapa (o **grafo**) es el espacio en el que el algoritmo A* usa para encontrar un camino entre dos posiciones. No es necesario que sea un mapa en el sentido literal de la palabra El mapa puede estar formado por cuadrados o hexágonos, puede ser un área tridimensional o puede tener una representación espacial de árbol. Lo importante a comprender es que el mapa es el espacio de búsqueda.

Nodos son las estructuras que representan posiciones en el mapa. El mapa puede ser una estructura independiente de los nodos. Los nodos almacenan información crítica para el algoritmo A* y también información posicional. De esta manera los nodos actúan como contadores para almacenar el progreso de la búsqueda del camino. Es importante saber que dos o más nodos pueden corresponder a la misma posición en el mapa (acceso por caminos diferentes).

Distancia (o **heurística**) es usada para determina la “aptitud” del nodo para ser explorado. Usaremos el termino distancia ya que centraremos la aplicación del algoritmo para mapas bidimensionales (en un entorno tridimensional).

Coste de un nodo quizá sea el término más difícil a definir. Usaremos una analogía para definirlo. Cuando hacemos viajes de larga distancia, tomamos muchos factores en cuenta (tiempo, energía, dinero, paisajes, etc.) esto afecta a que camino tomaremos. Los posibles caminos entre el inicio y el final llevan asociados costes y minimizar esos costes es el trabajo del algoritmo A*.

Pasaremos ahora a la teoría en la que rodea al algoritmo A*. El algoritmo A* recorre el mapa creando nodos que se corresponden con las diferente posiciones que va explorando. Hay que recordar que esos nodos permiten guardar el progreso de la búsqueda. Además de almacenar la posición del mapa dispone, entre otros, de tres atributos principales llamados comúnmente f, g y h en referencia a fitness (aptitud), goal (objetivo) y heuristic (heurística) respectivamente. Describámoslos más a fondo:

g es el coste de llegar desde el nodo inicial hasta ese nodo. Muchos caminos diferentes van desde el nodo inicial a esta localización del mapa, pero este nodo representa un único camino.

h es el coste estimado para llegar desde este nodo al nodo objetivo. Este nodo, como hemos comentado antes, representa la heurística utilizada y significa que es una aproximación del coste al final ya que realmente no conocemos ese coste (si lo conociéramos no nos haría falta buscar).

f es la suma de g y h. f representa nuestra mejor estimación para el camino que pasa a través de este nodo. Cuanto mas bajo sea el valor de f , creemos que mejor es el camino.

El propósito de f, g y h es cuantificar como de prometedor es el camino actual hasta ese nodo. El componente g es algo que podemos calcular perfectamente debido a que es el valor del camino hasta ese nodo. Ya que hemos explorado todos los nodos que preceden a ese nodo podemos conocer el valor exacto de g. Sin embargo, el componente h es totalmente diferente. Cuanto mejor sea nuestra estimación, más próximo f es el valor verdadero y más rápidamente el algoritmo A* encuentra el nodo final que menos esfuerzo.

Adicionalmente, A* mantiene dos listas, una lista de nodos abiertos y otra lista de nodos cerrados. La lista de nodos abiertos contiene los nodos que aun no han sido explorados y la lista de nodos cerrados contiene todos los nodos que ya han sido explorados. Un nodo es considerado como “explorado” si el algoritmo a procesado todos los nodos “hijos” conectados con este , es decir a calculado para cado nodo conectado con este el valor de f, g y h y han sido añadidos a lista de nodos abiertos para que sean explorados en un futuro.

Las listas de abiertos y cerrados son necesarias ya que los nodos no son únicos. Por ejemplo si empezamos en (0,0) y nos movemos a (0,1) un movimiento valido sigue siendo volver a (0,0).Por lo tanto hay que llevar cuenta de que nodos han sido procesados o no. Como hemos indicado antes, los nodos simplemente marcan el estado y progreso de la búsqueda.

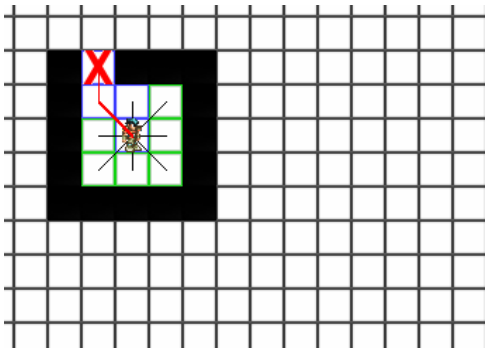
En búsqueda de caminos, esta distinción es importante ya que puede haber muchas vías diferentes para llegar al mismo punto. Por ejemplo, si un camino se divide en dos pero convergen más tarde, el algoritmo debe determinar que rama tomar.

Pasemos ahora a describir el algoritmo en pseudocódigo:

1. Sea P = el punto de inicio.
2. Asignar los valores de f, g y h a P
3. Añadir P a la lista de nodos abiertos. En este punto P es el único nodo en la lista de nodos abiertos.
4. Sea B = el mejor nodo (Best) de la lista de nodos abiertos, es decir el nodo que tiene el valor de f más bajo.
    a. Si B es el nodo objetivo entonces la búsqueda ha finalizado.
    b. Si la lista de nodos abierto es vacía no es posible encontrar un camino entre esos dos puntos

5. Sea C = un nodo valido conectado con B
    a. Asignar los valores de f, g, y h a C
    b. Comprobar si C esta en la lista de nodos abiertos o cerrados
        i. Si esta, comprobar si el camino es mas eficiente que el antiguo
            1. Si es así actualizar el camino
        ii. Sino añadir C a la lista de nodos abiertos
    c. Repetir el paso 5 para todos los hijos validos de B
6. Repetir desde el paso 4

Ejemplo simple del algoritmo:



Supongamos que la casilla marcada con la X sea la posición (0,1) y l posición central el inicio de la búsqueda (2,2).

Los valores iniciales son fáciles de calcular para el nodo inicial. g=0 ya que no se ha iniciado la búsqueda. Para calcular la estación de h usaremos la distancia matan, es decir la suma combinada de las diferencias entre las coordenadas verticales y horizontales (positivamente) del nodo actual y el final.

$$h = | dx – sx | + |dy – sy|$$

En nuestro caso: $h = |1-2|+|0-2| = 3$
Por lo tanto $f = g + h = 3$

Generamos los 8 hijos del nodo inicial. Su coste de g será g=1 (mover una casilla). El valor de h será diferente para cada uno pero se puede ver fácilmente que la casilla (1,1) es la mejor casilla (la más cercana al objetivo).

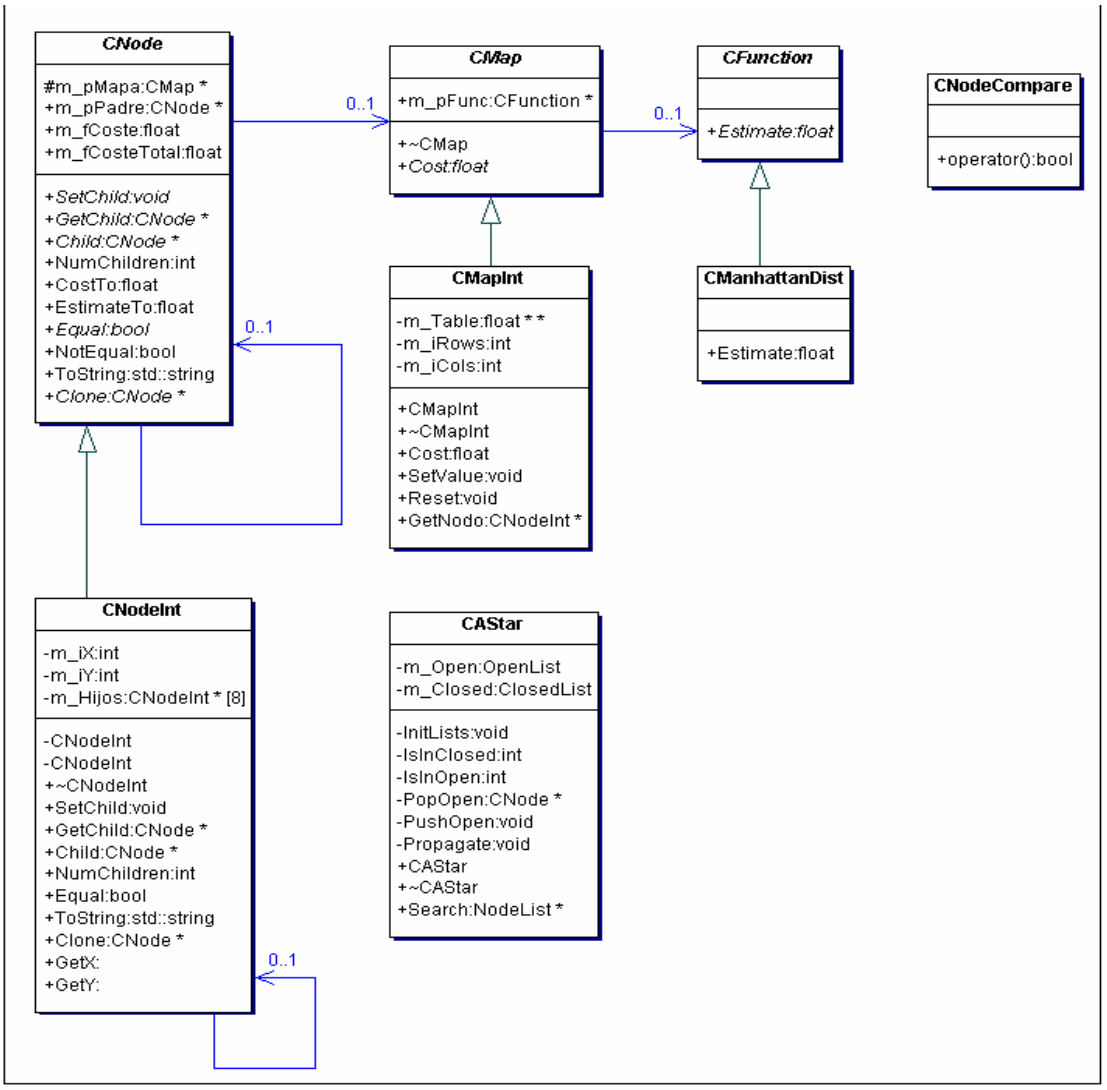
El nodo (1,1) tiene 4 hijos validos (1,0), (1,2), (2,1) y (2,2).Ahora tenemos que determinar que nodos están en la lista de nodos abiertos en la de cerrados y los nuevos nodos.

Cerrados: (2,2)
Abiertos: (1,2) (2,1)
Nuevos: (1,0)

Después de asignar los valores f, g y h es evidente que el nodo (1,0) tiene el mejor coste y que en la siguiente iteración llegamos al nodo final.

3.1.1.3. Implementación en el proyecto:

En el siguiente diagrama uml podemos observar la estructura de clases utilizadas para la implementación de algoritmo A*:



Código involucrado:

- CAStar.cpp
- CAStar.h
- CFunction.cpp
- CFunction.h
- CManhattanDist.cpp
- CManhattanDist.h
- CMap.cpp
- CMap.h
- CMapInt.cpp
- CMapInt.h
- CNode.cpp
- CNode.h
- CNodeInt.cpp
- CNodeInt.h

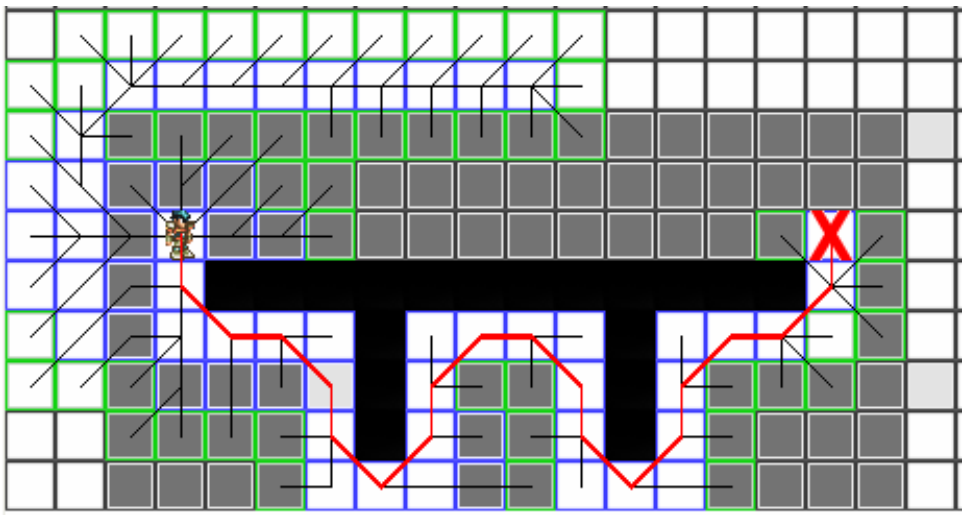
Funciones destacadas:

```
CAStart::CAStar();

CAStar::NodeList* Search(CNode* pStart, CNode* pGoal);

void CNode::SetChild(int i,CNode* pNode) = 0;

CNode* CNode::GetChild
```

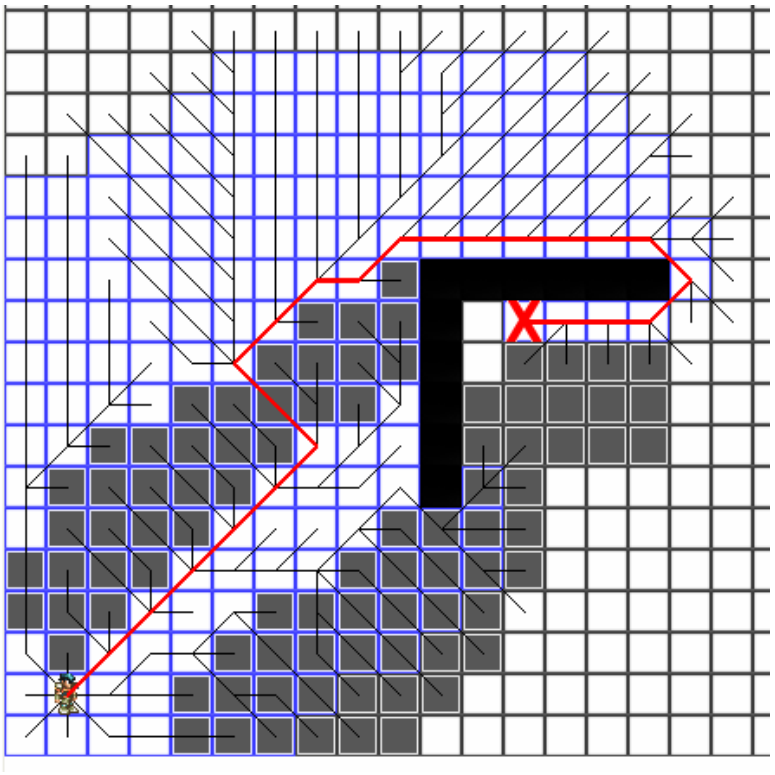
Para más información ver el código y la documentación adjunta a los archivos.

3.1.1.4. Imágenes:

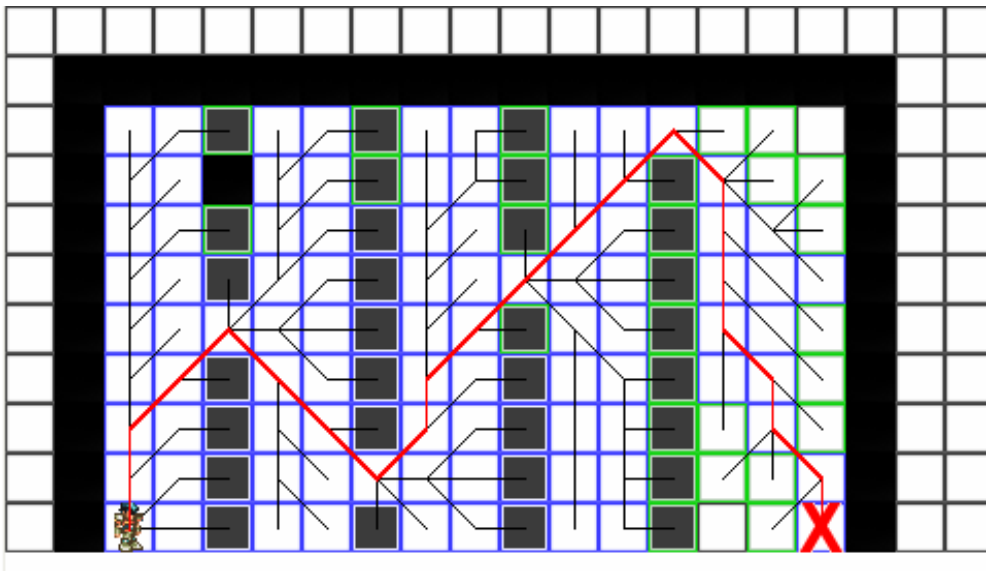
Algunas imágenes del algoritmo en funcionamiento y los caminos obtenidos:



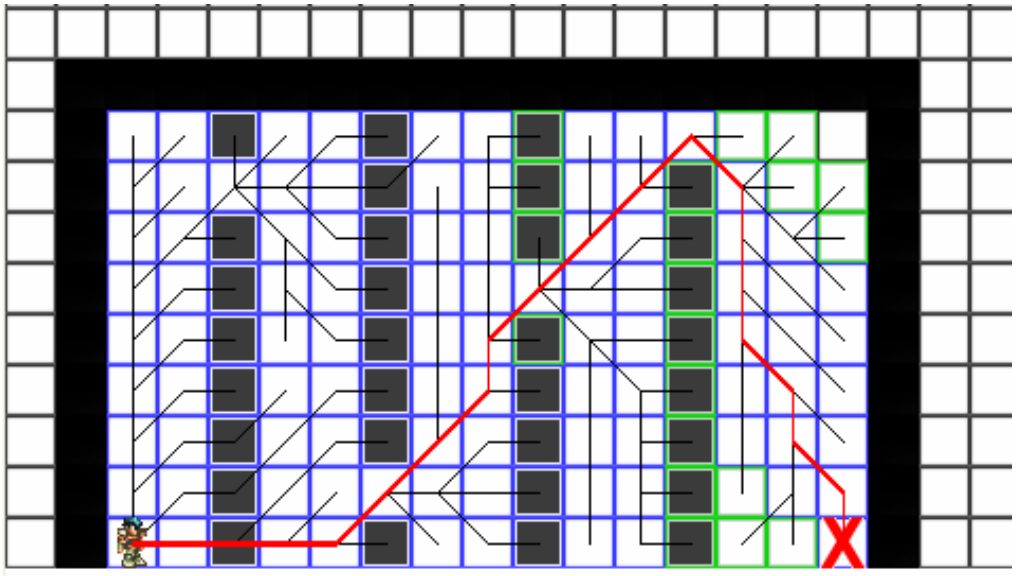
Se puede ver como sigue el pasillo en vez de ir directamente en línea recta.



El mapa simula una carretera cortada. El camino tiene que rodearla hasta encontrar el objetivo. Gran cantidad de nodos abiertos



Se puede “obligar” a que pase por ciertas zonas al personaje colocando.



Aunque como se puede ver no necesariamente debe pasar por las zonas “más fáciles”

3.1.2 Algoritmo de Floyd

3.1.2.1 Qué es

Sea G = <N,A> un grafo dirigido; N es el conjunto de nodos y A el de aristas. Toda arista tiene asociada una longitud no negativa. El algoritmo de Floyd es un algoritmo utilizado para calcular la longitud del camino más corto entre cada par de nodos de un grafo.

3.1.2.2 Cómo funciona

Este algoritmo utiliza técnicas de tabulación y de programación dinámica.

Se basa en el principio de optimalidad: supongamos que los nodos de G están numerados desde 1 hasta n, así que N = {1,….n} y supongamos que hay una matriz L que da las longitudes de las aristas, con L[i,i] = 0 para i=1,2,…,n, L[i,j]>=0 para todo i y j, y L[i,j] = 8 si no existe la arista (i,j). Si k es un nodo del camino mínimo entre i y j, entonces la parte del camino que va desde i hasta k, y la parte del camino que va desde k hasta j debe ser óptimo también.

En primer lugar, se construye una matriz, llamémosla D, que da la longitud del camino más corto entre un par de nodos. El algoritmo da a D el valor inicial L, esto es, entre nodos. Luego efectúa n iteraciones. Después de la iteración k, D da la longitud entre los caminos más cortos que usan sólo los nodos [1,2,…, k] como nodos intermedios. Al cabo de n iteraciones, D nos da por tanto la longitud de los caminos más cortos que usen algunos de los nodos de N como nodo intermedio, que es el resultado buscado. En la iteración k, el algoritmo debe comprobar que para cada par de nodos (i,j) si existe o no un camino que vaya de i a j pasando por el nodo k, y que sea mejor que el camino óptimo actual que pasa sólo por los nodos [1,2,…,k-1]. Si D_k representa la matriz D después de la k-ésima iteración (de modo que D_0 = L), entonces la comprobación necesaria tiene la siguiente forma:

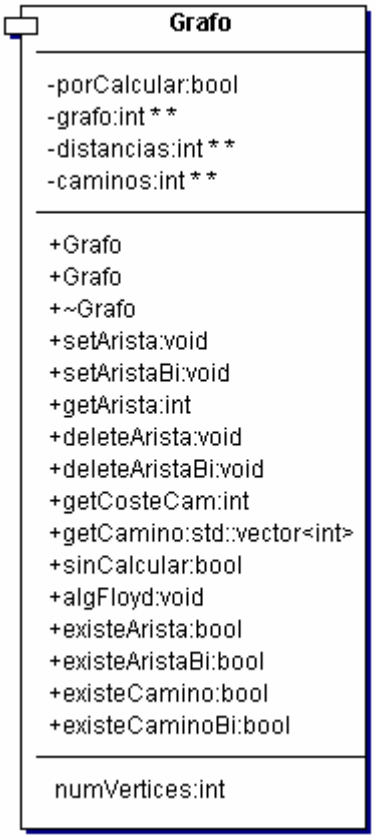
$$D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$$

Estamos usando el principio de optimalidad para calcular el camino más corto que va desde i hasta j pasando por k. Tácitamente, hemos usado el hecho de que un camino óptimo que pase por k no visitará k dos veces,

En la k-ésima iteración, los valores de la k-ésima fila y de la k-ésima columna de D no cambian, porque D[k,k] es siempre 0. Por tanto, no hace falta guardar estos valores al actualizar D. Esto nos permite tener una única matriz D n x n, mientras que a primera vista pudiera parecer necesario disponer de dos de estas matrices, una que contendría los valores de D_{k-1} y otro los de D_k, o incluso una matriz de n x n x n.

3.1.2.3 Implementación en el proyecto

El diagrama de clases utilizadas es el siguiente:



Código involucrado:

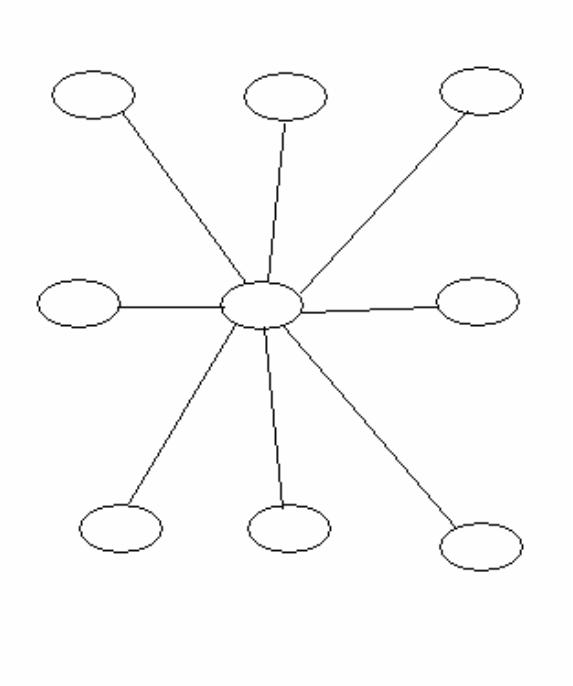
- Grafo.h
- Grafo.cpp

Funciones destacables:

```
Grafo(int numNodos);

void setArista(int i, int j, int coste);

void setAristaBi(int i, int j, int coste);

void algFloyd();

int getCosteCam(int i, int j);
```

Para más información, consultar el código y su documentación.

En el proyecto, este algoritmo se ha utilizado para calcular el camino entre diferentes zonas del barrio de clases y objetos. Se dispone de un grafo de conectividad muy simple, con estructura de estrella, con la calle como centro.



3.2 Carga de modelos de personajes:

3.2.1. Qué es:

La carga de modelos de personajes nos permite la ubicación en memoria de toda la información necesaria para un modelo determinado (texturas, vértices, animaciones, etc.) mediante la adaptación de las estructuras predefinidas al motor gráfico utilizado.

Debido a la gran cantidad de formatos de modelos 3D existentes y debido, sobre todo, a la proliferación de modelos diseñados gratuitamente por gente en Internet se a optado por utilizar un formato conocido, que cumpla con las necesidades de animación de la aplicación y que tenga disponibles modelos suficientes para nuestros cometidos.

El formato elegido para los modelos es el del juego Half-Life cuya extensión es ".mdl" y es muy conocido entre la comunidad aficionada a los juegos de ordenador. Este formato presenta todas las características que necesitamos y , lo más importante, tiene documentación disponible en la red para llevar acabo la adaptación.

3.2.2. Como funciona:

La mayoría de los formatos de modelado 3D tienen una estructura parecida. Siempre existen pequeñas diferencias de cómo se manejan las animaciones o como se almacenan las texturas y referencias a los vértices, pero en el fondo son muy parecidas.

En general existe una cabecera que almacena información referente al número de vértices, normales, triángulos, texturas, huesos, joints, sonidos, submodelos, etc., información referente a características generales, tales como nombre del modelo, posición inicial e índices al comienzo del los arrays con toda esa información.

Cada estructura descrita dispone a su vez de una cabecera que indica como se organiza esa información en la dirección de memoria donde apunta el índice.

Así podemos imaginar que queremos simplemente pintar los triángulos del modelo sin texturas ni animaciones los pasos a seguir serían los siguientes:

- Leemos el modelo de disco.
- Accedemos a la cabecera y buscamos la dirección donde comienza la información del modelo.
- Accedemos a la información del modelo y buscamos la dirección donde almacena los polígonos (normalmente triángulos) que dan forma a la malla 3D del modelo.

- Accedemos a los polígonos y leemos los índices del array de vértices y normales que componen el polígono.
- Buscamos la dirección de memoria del array de vértices y accedemos con los índices obtenido al array. Ídem para las normales.
- Con la información del valor de cada vértice y cada normal pintamos los triángulos.

La información de esta manera se almacena jerárquicamente y no necesita guardar valores de vértices, normales etc. repetidos.

3.2.3. Implementación en el proyecto:

La empresa creadora del juego Half Life (Valve LLC.) proporciona el código necesario para dibujar y animar los modelos mediante el uso directo de opengl.

Si bien este código no podemos utilizarlo directamente, nos permite comprender la forma en la que esta estructurado el modelo y el uso de las cabeceras del modelo predefinidas.

Se ha hecho una adaptación de las funciones de pintado del modelo que nos permite devolver la información que necesita el TriMesh para ser pintado por el motor. Estas funciones nos devuelven los vértices a pintar con su correspondiente normal, color y coordenada de textura.

Código involucrado:

- .\src\Engine\math.cpp
- .\src\Engine\mathlib.h
- .\src\Engine\mdlviewer.cpp
- .\src\Engine\mdlviewer.h
- .\src\Engine\MgcExMeshHL.cpp
- .\src\Engine\MgcExMeshHL.h
- .\src\Engine\studio.h
- .\src\Engine\studio_render.cpp
- .\src\Engine\studio_utils.cpp

Cabe destacar las funciones:

StudioModel

```
int                                 getVertexQuantity();
vec3_t*                             getVertex();
int                                 getNormalsQuantity();
vec3_t*                             getNormals();
int                                 getTriangQuantity();
int                                 getTriangQuantity2();
short*                              getTriang();
void                                SetupModel ( int bodypart );
std::vector<float*>                 getPoints( );
std::vector<float*>                 getColors( );
std::vector<float*>                 getAll( );
std::vector<float>                  getS( );
std::vector<float>                  getT( );
std::vector<float*>                 getSTN( );
byte*          getTexture( mstudiotexture_t *ptexture, byte *data, byte *pal );
```

MeshHL

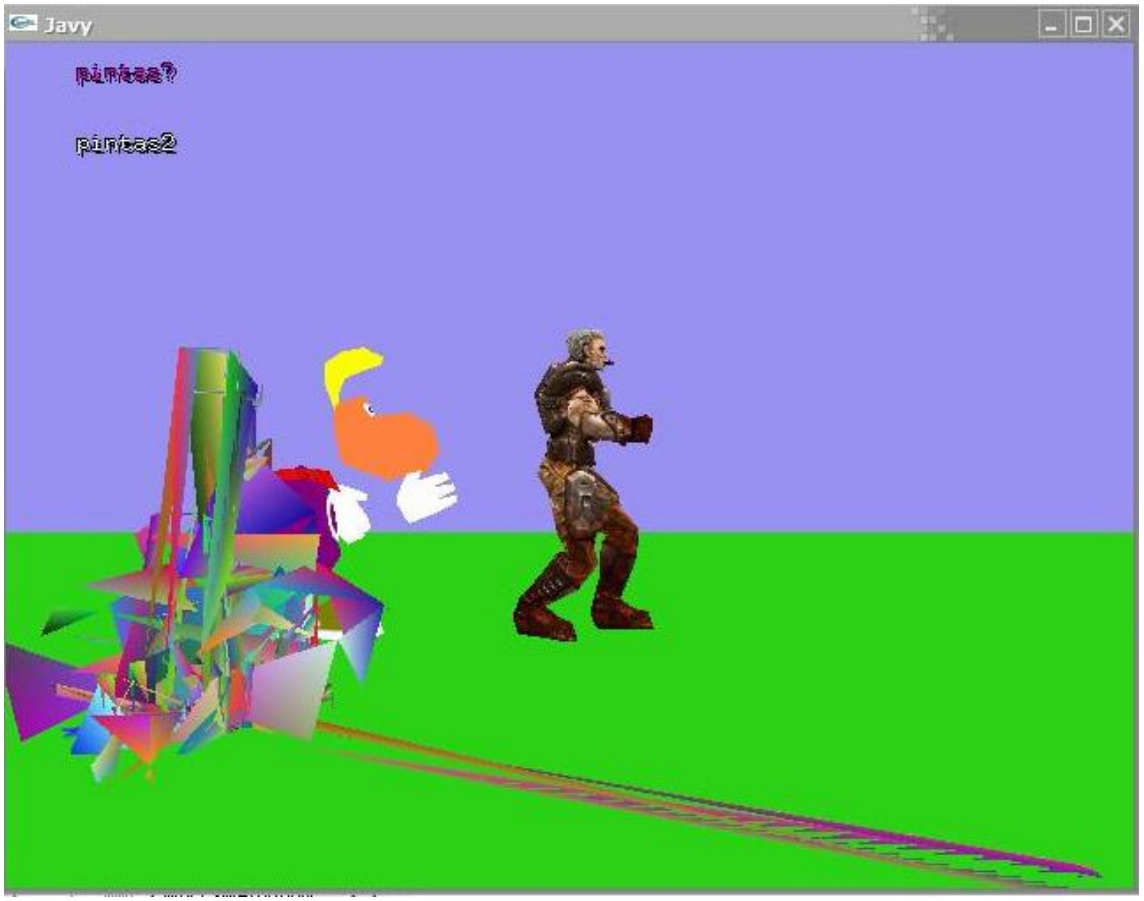
```
MeshHL(char *modelname);
virtual void Draw (Mgc::Renderer& rkRenderer);
void updatePoints(float fAppTime);
```

El peso mas importante recae sobre la constructora de MeshHL que hace gran parte de la adaptación al código del motor.

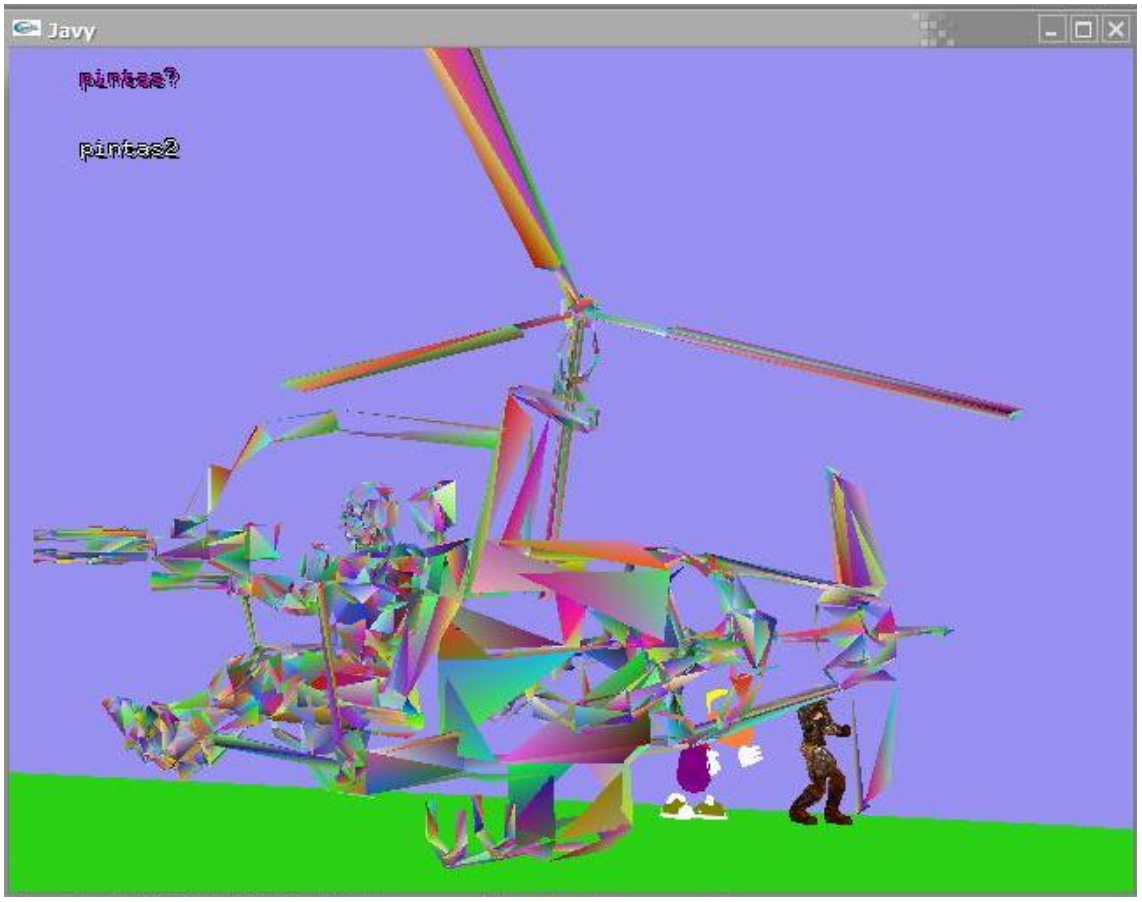
3.2.4. Imágenes de modelos:

Las siguientes imágenes muestran la evolución gráfica que ha tenido la carga de modelos en el proyecto. Se puede observar los avances según se iban resolviendo problemas en la adaptación de la carga de modelos al motor utilizado. En las imágenes se verán 2 modelos cargados del “Quake 3” cuya carga ya la implementaba el motor pero que no satisfacían los requisitos de animación propuestos.

Como primer objetivo se planteo la carga correcta de la malla de triángulos del personaje, más tarde se intentaría la carga correcta de texturas y multitexturas por personaje.



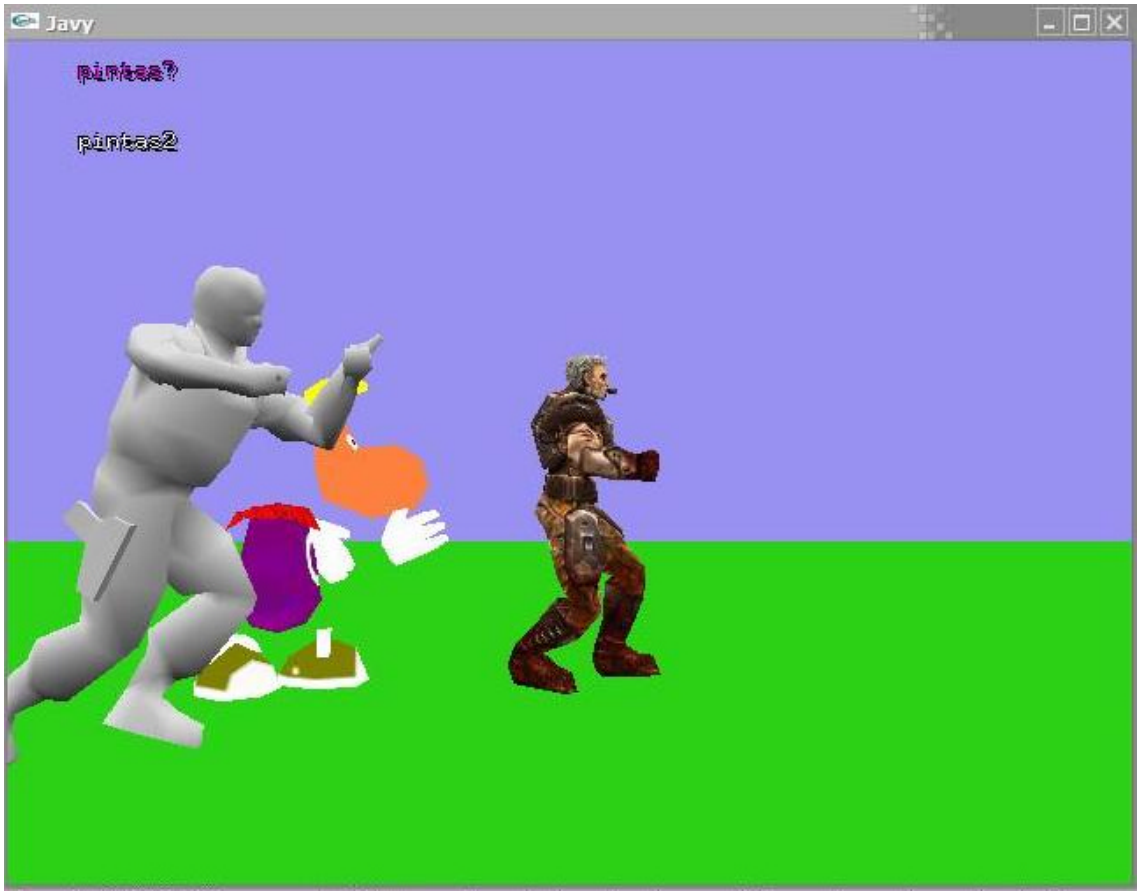
Primer intento de carga de un personaje, el acceso a la información de los triángulos era totalmente incorrecto y el acceso a las normales también. Los polígonos tenían colores aleatorios en cada vértice para poder diferenciarlos y poder hacer un debug más visual.



Tras subsanar los fallos de acceso a memoria y a los arrays que almacenan los vértices de las mallas, podemos observar que el modelo va adoptando ciertas formas mas reconocibles.

En este caso se está intentando cargar un modelo que representa un mini helicóptero con su conductor montado en él.

Se puede observar que el modelo presenta en todo su extensión una serie de “huecos” donde deberían ir triángulos. Esta dificultad presento serios problemas para solventarla, pero una lectura en profundidad del código de Valve y de opengl nos permitió avanzar.



Podemos observar el gran avance entre esta foto y la anterior, sin duda presenta un aspecto mas definido y un sombreado uniforme que redondea el modelo.

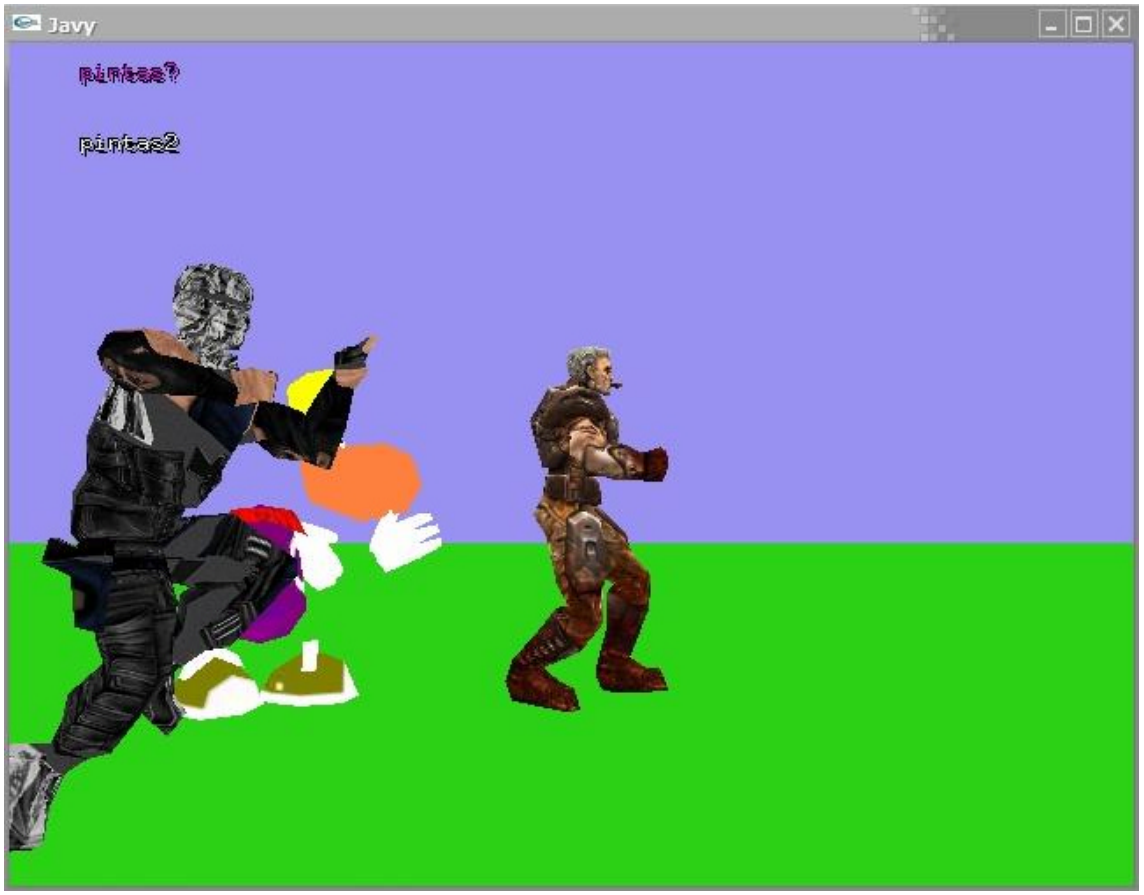
El problema de los “huecos” anterior estaba relacionado con la forma de pinta GL_TRIANGLE_STRIP y GL_TRIANGLE_FAN de opengl en el modelo.

La estructura de TriMesh del motor gráfico utilizado no permitía estas dos formas de pintado de triángulos (la primera, pintado de triángulos contiguos compartiendo dos vértices y la segunda, pintado de triángulos compartiendo todos un mismo vértice).

Para resolver este problema se opto por repetir vértices y así poder pintar los triángulos como si fueran disjuntos. El problema estaba en que las estructuras alternar el orden de los vértices de los triángulos por lo que el cálculo de las normales que corre a cargo del motor se realizaba incorrectamente.

Debido a este cálculo incorrecto el motor no pintaba los triángulos con la normal incorrecta ya que esta apuntaba hacia el lado contrario al correcto y consideraba que no se debería ver ese polígono.

Se puede observar que también se han asignado los colores correctos a cada vértice del triángulo así el modelo presenta un aspecto más antropomórfico

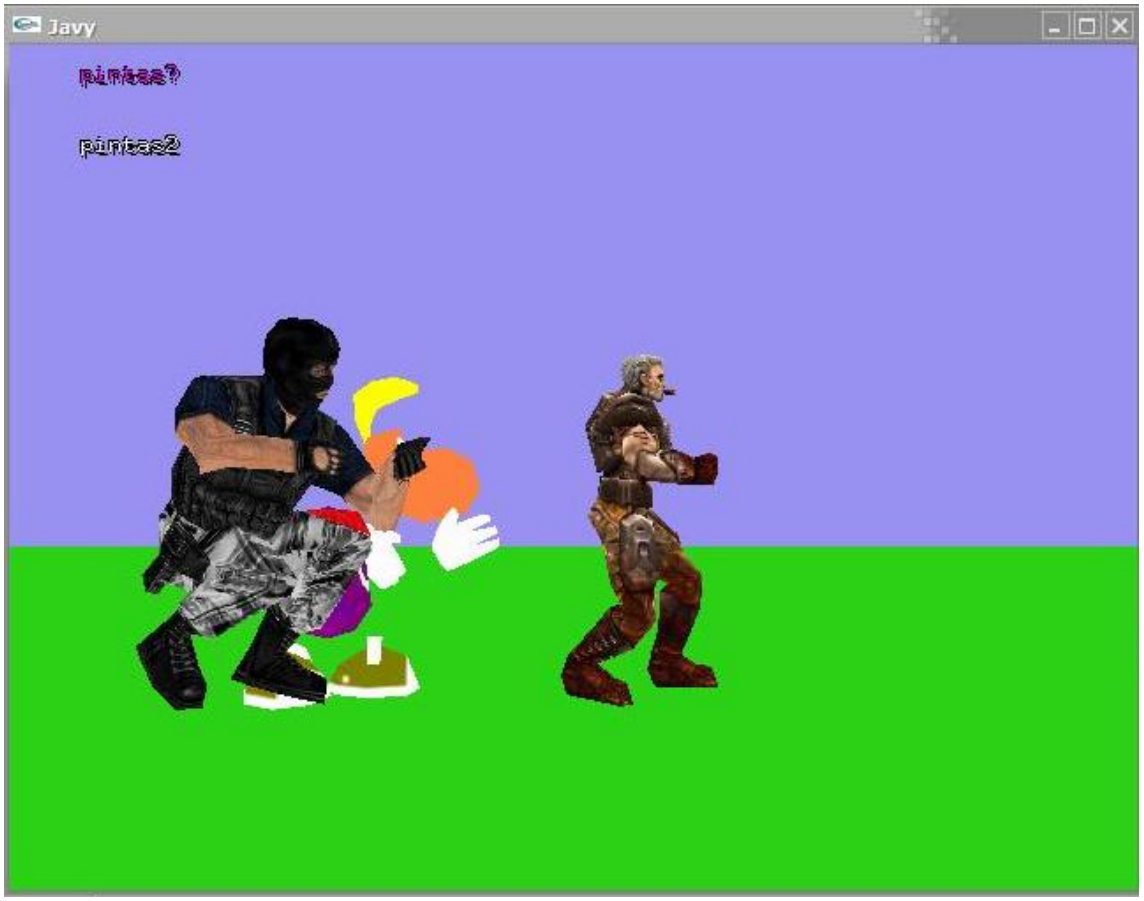


Entramos en la segunda fase descrita, la carga de texturas para el personaje. Primer intento de carga de texturas, el acceso a las texturas internas del modelo se realizaba incorrectamente por lo que se decidió cargar una textura externa en formato JPG.

La carga se realizaba con la clase ImageLoader del motor y se asociaba al único TriMesh del modelo utilizado.

Las coordenadas de textura eran incorrectas por lo que según se ve en la imagen las diferentes partes del cuerpo salían más o menos descolocadas. La zona de la cabeza corresponde a los pantalones por ejemplo.

Más tarde se observo que esto era debido a que ImageLoader cargaba al revés verticalmente las texturas. Pero decidimos que había que conseguir cargar las texturas internas del modelo.



Podemos observar el resultado final tras solventar todos los problemas de carga acceso y carga de texturas.

En el motor gráfico utilizado (WildMagic) sólo es posible asociar una única textura por malla de triángulos TriMesh. Esto que a priori es una limitación muy grande se puede solucionar generando varios TriMesh diferentes aunque introduce pequeños problemas para la animación.

3.3 Animación esquelética:

3.3.1. Qué es:

La animación esquelética es una técnica usada para animar modelos de caracteres. Un esqueleto es asociado al modelo del carácter el carácter se convertirá en la piel del esqueleto y la animación se encargará de hacer transformaciones sobre el esqueleto dando un movimiento realista al carácter.

Desde hace varios años, la mayoría de los juegos que representan un entorno 3D a optado por implementar un motor de animación esquelética mas o menos complejo que se adaptara a sus requisitos. Así juegos como Half Life, Serius Sam, Unreal Torunament, The Sims , … han optado por este tipo de animación para sus modelos llegando a alcanzar grandes cotas de realismo. Otros juegos como Quake 1, 2, 3, Return To Castle Of Wolfstein han optado por la animación mediante interpolación de keyframes.

3.3.2. Cómo funciona:

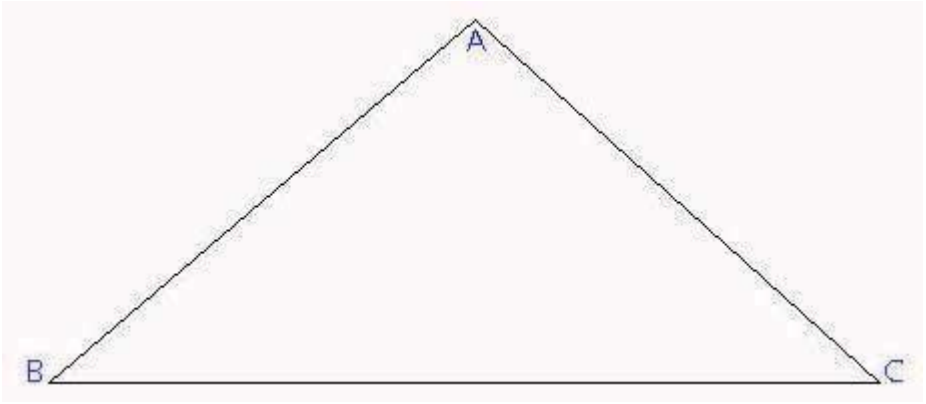
La idea básica puede resultar bastante familiar, un conjunto de huesos (bones) son creados y añadidos a la malla de triángulos del modelo (que en este caso se podría hacer un símil entre la malla de triángulos y los “músculos” del personaje).

Cada vértice de la malla se corresponde con un hueso del modelo. También puede darse la posibilidad de que no tenga correspondencia con ningún hueso entonces ese vértice nunca se moverá (poco utilizado) o que tenga correspondencia con mas de un hueso por lo que podrá describir distintos movimientos e incluso combinaciones de estos. En la práctica, normalmente, cada vértice suele tener correspondencia con un solo hueso ya que los sistemas de animación esquelética con correspondencia con dos o más huesos suelen ser demasiados complejos, aunque de un gran realismo.

Cuando los huesos se mueven los vértices que le corresponden se mueven de la misma manera que lo haga el hueso manteniendo su posición relativa a él.

Cada hueso tiene asociado “joints” (termino que podríamos traducir como juntura) que almacenan la información relativa a las transformaciones del hueso (matrices de rotación traslación). Estos joints forman una red jerárquica en la cual el movimiento de uno afecta directamente a todos los joints inferiores.

Un ejemplo simple que ilustra está teoría, podría ser el siguiente:

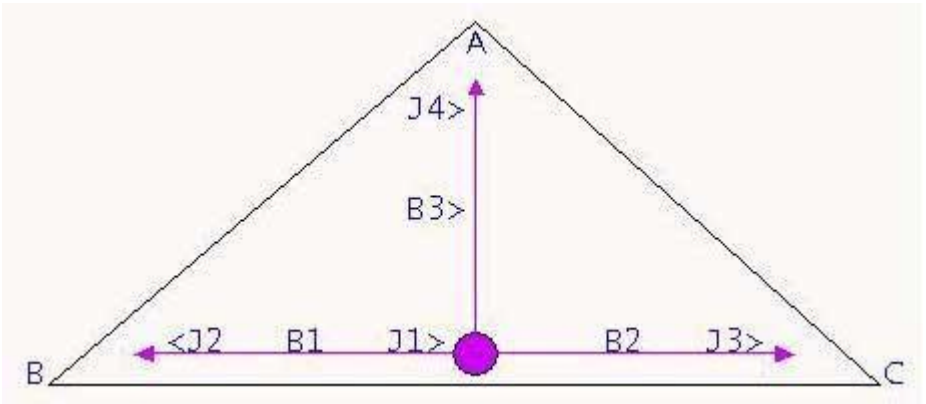


** Ejemplo inspirado en la leccion 03 de Nehe Productions*

Supongamos que tenemos un triangulo formado por los puntos A B C que forman un único plano en un entorno 3D. Ahora supongamos que queremos que el punto A descienda un poco y los puntos B y C vayan hacia fuera del plano (es decir dirección hacia el lector). Además queremos que el triangulo gire sobre si mismo al mismo tiempo.

¿Como se podría realizar todas estas transformaciones? Una opción sería calcular y almacenar todas las coordenadas para cada punto en cada frame (fotograma) y después simplemente colocar los puntos donde corresponda en cada frame, pero esto es lento, estático y poco elegante. Aun así hay que calcular las rotaciones y puede que sea bastante difícil.

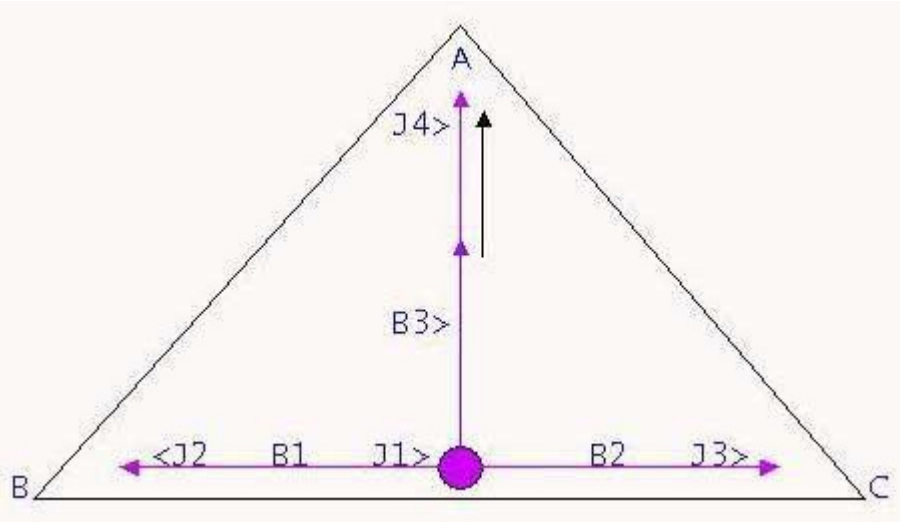
Veamos como se podría hacer con animación esquelética.



Para los movimientos descritos necesitaremos 3 (B1 B2 B3) huesos y 4 joints (J1 J2 J3 J4). Podemos observar que todos los huesos y los joints J2 J3 J4 (joints finales) nacen del joint J1. Esto esencialmente quiere decir que cuando se mueve J1 todos sus huesos hijos y sus joints finales se mueven junto a el (si va hacia arriba todo se mueve hacia arriba por ejemplo).

En un sentido similar si un joint hijo se mueve y no tiene hijos por debajo de él, solamente él es afectado por ese movimiento. Por ejemplo, si movemos J4 solamente este se moverá.

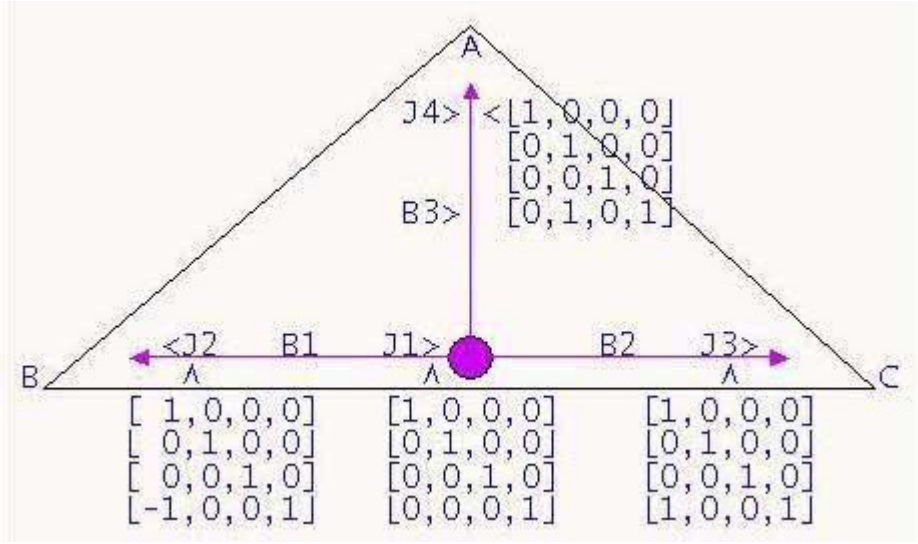
Como hemos comentado anteriormente, cada vértice suele estar asociado a un hueso del modelo. En nuestro caso tendremos las asociaciones A-J4 B-J2 C-J3. Cada uno controla el movimiento de cada vértice y J1 controlaría el movimiento de toda la figura.



Movimiento único de A-J4

Todos los movimientos son realizados por un complejo sistema de matrices, frames, keyframes y animaciones. Además, cada joint contiene información de traslación y rotación (matrices relativa y absoluta).

Los keyframes son las posturas fundamentales del esqueleto con las que se define la secuencia de la animación. Para generar entre dos keyframes las sucesivas posturas del modelo se utiliza interpolación a través de quaternions.



Cada vértice adjunto a un joint es transformado por la matriz y salvado en una versión temporal del modelo. Esta versión deformada del modelo es mostrada por pantalla.

Así podemos ver que es un sistema efectivo, rápido y muy potente para definir movimientos independientes (aunque complejo de implementar).

3.3.3. Implementación en el proyecto:

En el proyecto se ha optado por utilizar modelos del juego Half-Life, básicamente por dos razones importantes:

1- Es un juego muy conocido y extendido que tiene a su disposición una gran cantidad de modelos en Internet y en sus diferentes mods.
2- Existe código de la empresa creadora (Valve LLC.) que indica como cargar los modelos y animarlos

La implementación de un motor de animación esquelética es una tarea compleja y que requiere de grandes conocimientos matemáticos (interpolación de keyframes mediante quaternions) así como de robótica, cinemática inversa etc. Constituiría un gran esfuerzo para nuestros objetivos la realización desde cero de un motor de esta categoría.

Gracias al código de la empresa creadora del Half-Life (Valve LLC) disponible en Internet la tarea simplifica de animar los modelos se simplifica a la mitad.

Mediante la adaptación de los algoritmos de animación y estructuras de Valve es posible realizar una integración dentro de nuestra aplicación de la animación esquelética.

Código involucrado:

- .\src\Engine\math.cpp
- .\src\Engine\mathlib.h
- .\src\Engine\mdlviewer.cpp
- .\src\Engine\mdlviewer.h
- .\src\Engine\MgcExMeshHL.cpp
- .\src\Engine\MgcExMeshHL.h
- .\src\Engine\studio.h
- .\src\Engine\studio_render.cpp
- .\src\Engine\studio_utils.cpp

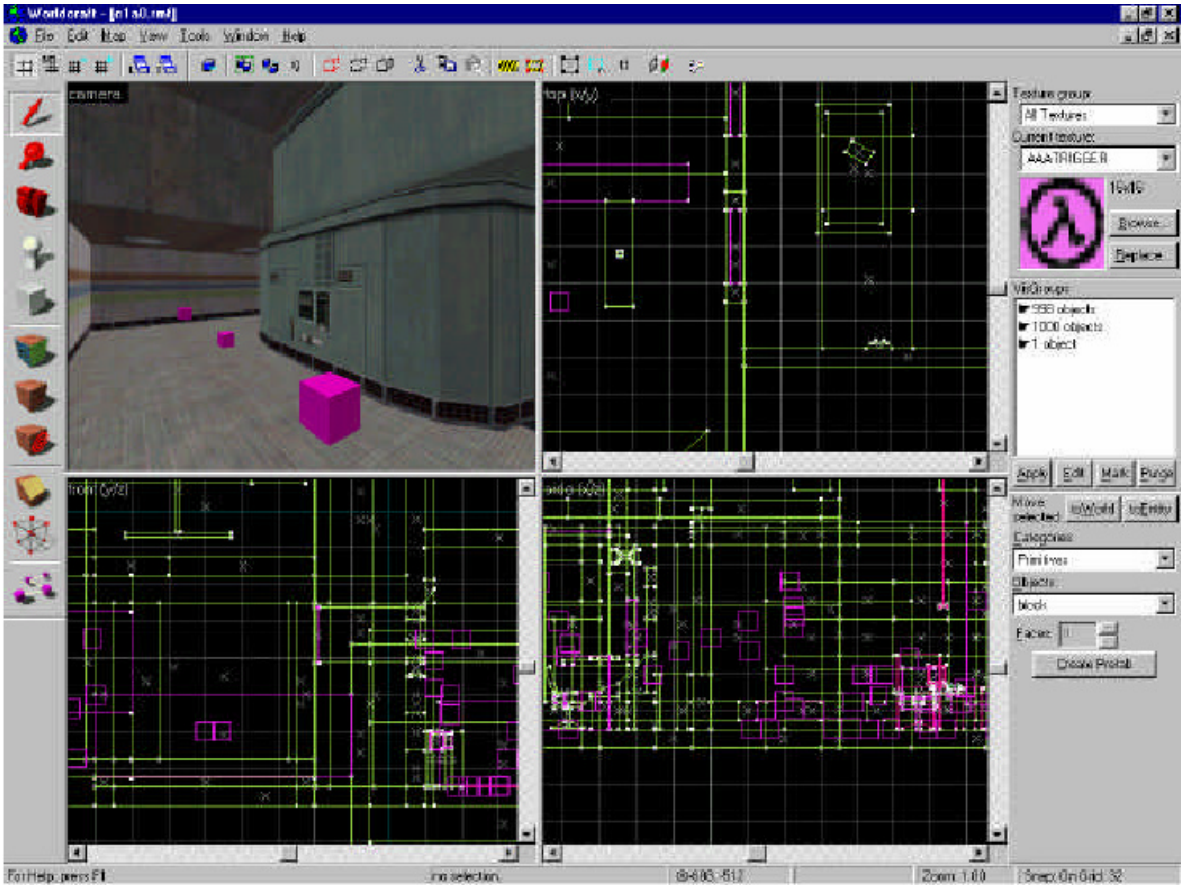
Funciones destacadas:

```
void StudioModel::SetUpBones ( void );

void StudioModel::SetupLighting ( );

void StudioModel::SetupModel ( int bodypart );

void StudioModel::DrawModel( );
```

3.3.4. Imágenes:

3.4 Carga de escenarios

3.4.1 El editor de mapas WorldCraft



Los niveles de Worldcraft están formados por dos tipos principales de objetos, “brushes” y entidades. Los primeros son geometría de nivel (o polígonos) que se compilan luego en un árbol BSP. Son estáticos, no se mueven ni cambian. Por eso son geniales para suelos, paredes, techos, etc. Las entidades son los disparadores invisibles de un nivel, como luces y sonidos de ambiente. No contienen ningún tipo de geometría. Entre “brushes” y entidades están las entidades sólidas, objetos que contienen geometría pero que no forman parte del árbol BSP. Éstas típicamente incluyen estatuas, árboles, estanterías, mesas, botones y puertas. Podemos pensar que algunos de estos objetos no se mueven o tienen alguna función especial, excepto ser decorativos. La única razón por la que son entidades sólidas es porque sería estúpido tenerlas como parte del árbol BSP del nivel (especialmente cuando se calculan los PVS). Por ejemplo, en una estatua con muchos polígonos, es obvio tratarla como entidad sólida, porque únicamente aumentaría el árbol BSP (por no hablar del PVS). Las entidades sólidas no se usan sólo con objetos pequeños con muchos polígonos, sino que también para enlazar una función a una

geometría, como una puerta. Los botones, niveles, palancas, puertas y elevadores son normalmente entidades sólidas.

Así que para resumir:

- Paredes, suelos, techos y terrenos deberían ser “brushes”.
- Sonidos de ambiente, objetivos, y sistemas de partículas deberían ser entidades.
- Puertas, botones, objetos con muchos polígonos y elevadores deberían ser entidades sólidas.

Hemos usado Worldcraft, porque para nuestras necesidades no creíamos conveniente reinventar la rueda, existiendo ya un editor de niveles con la funcionalidad necesaria para nuestro proyecto.

Es posible personalizar el editor. El primer paso para ello es comprender el uso de las entidades. Los juegos pueden describir sus entidades al editor Worldcraft mediante el formato de fichero .FGD. Este formato de fichero proporciona una forma orientada a objetos para describir entidades. Todo es una clase. Aun así, no hay métodos, sólo atributos.

Los datos se definen usando la sintaxis:

Nombre(tipo) :”Etiqueta” : valor_por_defecto

Donde nombre es el nombre del capo, tipo representa uno de los siguientes tipos de datos: **integer**, **string** y **color255**. La etiqueta es lo que Worldcraft muestra en la ventana Properties (no muestra el nombre del campo).

Valor_por_defecto es, obviamente, el valor por defecto.

Hay dos tipos más de datos que puede tener un campo. El primero es **choices**, que básicamente es una list-box. Se define así:

Nombre(choices): valor_por_defecto =
[
valor1: “Etiqueta 1”
…
valorN: “Etiqueta N”
]

En este caso, si se elige Etiqueta 1, el valor del campo sería valor1.

El último tipo de datos es **flags.** No son campos de datos separados, sino un grupo. Cada flan tiene un valor de máscara de bits que le dice a Worldcraft que bit representa este flan. Estos datos se definen tal que así:

Nombre(flags) =

```
[
bit: “Etiqueta”:valor_por_defecto
…
bit: “Etiqueta”:valor_por_defecto
]
```

bit es la máscara de bits (1,2,4,8,16,32,64,128). Etiqueta es lo que Worldcraft muestra y valor_por_defecto (0 ó 1) le dice a Worldcraft cuál es el estado por defecto del flan. Cada entidad puede tener sólo un campo flags.

La sintaxis para definir una clase es:

```
@ClassType base(baseclass1, baseclass2, …, baseclassX) classname = “Descripción”
[
]
```

Descripción es lo que muestra Worldcraft. Classname es el nombre de la clase. Todas las clases de las que deriva la nuestra están dentro de la función **base()**. Por último, ClassType puede ser de tres tipos de clases diferentes: **BaseClass**, **PointClass** y **SolidClass**.

BaseClass describe uan clase que no se puede instanciar directamente. Otra clase debe extenderla. Es útilo para hacer propiedades estándar, como ID’s o nombre. Por ejemplo:

```
@BaseClass = target_owner
[
target_name(string) : “Target name:”
]
```

Esta clase podría ser extendida por cualquier objeto que quisiera tener un objetivo. Los objetos BaseClass no tienen ninguna descripción textual que mostrar en el editor, porque no se pueden instanciar directamente.

PointClass se usa para hacer entidades. Puede heredar una clase base. Estos objetos sólo tienen un origen (su posición), pero no tienen “brushes”. Éstas se usan para luces, objetivos, sistemas de partículas y sonidos de ambiente. Por ejemplo:

PointClass is used to make entities. A PointClass can inherit a base class. PointClass objects only have
an origin (their position), but no brush(es). These are used for lights, targets, particle systems, and
ambient sounds. For example:

```
@PointClass base(target_owner) size(x0 y0 z0, x1 y1 z1) = light_directional : “Directional Light”
[
color(color255) : “Color (R G B)”
spawnflags(flags) =
[
1 : “Initially on” : 1
]
```

]

Esta entidad luz direccional tiene un objetivo, un color de luz y usa un flag que determina si está inicialmente encendida. La función **size()** se usa para decirle a Worldcraft cómo de grande debería ser la caja mostrada en la ventana del editor.

SolidClass se usa para hacer entidades sólidas. Puede heredar de una clase base. Estas entidades a menudo tienen prefijos “func_”. Los objetos SolidClass tienen un origen (su posición), también “brushes”. Éstas se usan para puertas, botones, elevadores, y otras entidades sólidas controlables. Por ejemplo:

```
@SolidClass base(target_owner) = func_button : “Button”
[
reset_time(integer) : “Reset time (sec)” : 15
activate(choices) : “Activation sound” : 0 =
[
0: “Click”
1: “Hum”
2: “Chunk”
3: “Activated”
4: “Beep”
]
]
```

Este botón tiene un objetivo, un tiempo de reinicio y un sonido que se ejecuta cuando se activa el botón.

Es importante comprender que todo lo que se define en una entidad es tan sólo datos; nada ocurre en realidad. Nuestro juego tiene que procesar datos, cambiar su estado y enviar eventos. Por ejemplo, el objetivo de un botón no se activará sólo porque lo diga la descripción de la entidad, nuestro programa tiene que tener código que lo haga. Por último, recalcar que podemos hacer cosas avanzadas como darle a entidades PointClass iconos que se muestran en vez de cajas.

3.4.2 Archivos MAP

El formato MAP es ampliamente usado, y él único aceptado por los compiladores que efectúan la fase de construcción de geometría sólida (CSG). Los archivos MAP son creados por el editor WorldCraft y son la especificación "sin compilar" de los mapas para el Half-Life

La mejor forma de comprender su formato es estudiar un ejemplo. Veamos uno:

```
{
"classname" "worldspawn"
"mapversion" "220"
"wad" "\games\half-life\cstrike\cstrike.wad;\games\half-life\valve\halflife.wad"
{
( 0 64 64 ) ( 64 64 64 ) ( 64 0 64 ) BCRATE02 [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 0 0 0 ) ( 64 0 0 ) ( 64 64 0 ) BCRATE02 [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 0 64 64 ) ( 0 0 64 ) ( 0 0 0 ) BCRATE02 [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 64 64 0 ) ( 64 0 0 ) ( 64 0 64 ) BCRATE02 [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 64 64 64 ) ( 0 64 64 ) ( 0 64 0 ) BCRATE02 [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 64 0 0 ) ( 0 0 0 ) ( 0 0 64 ) BCRATE02 [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
}
}
```

Las llaves se colocan alrededor de cada entidad, de ahí que haya una al comienzo del fichero. Todo lo que hay en un fichero MAP pertenece a una entidad. Las entidades obvias son puertas, luces, posiciones de inicio de jugadores, etc. Sin embargo, incluso la geometría básica del nivel, que parece que no forma parte de una entidad es, de hecho, parte de una entidad. Cada entidad tiene dos partes: propiedades y "brushes".

Las propiedades consisten en un nombre, que nos dice que tipo de propiedad es, y un valor. Las propiedades nos ayudan a almacenar información no geométrica sobre un objeto. Esto podría incluir aspectos tales como la energía de un enemigo, o el nombre de fichero de un sonido que se utiliza cuando algo se activa.

Las "brushes" están encerradas entre llaves. Representan la geometría real del nivel. Esto incluye paredes, suelos, mesas, etc. (todo lo que consista en polígonos). Cada una consta de cuatro o más caras. Una cara es un plano, un nombre de textura e información de textura.

Hay una propiedad que debe poseer toda entidad: *classname*. Esta propiedad especifica qué tipo de entidad se está describiendo. Ejemplos de este caso serían: *"classname" "worldspawn"*, *"classname" "light"*, o *"classname" "button"*. Es la primera propiedad de cada entidad.

Worldspawn es una entidad que debe estar presente en todo archivo MAP. Es la primera entidad que aparece en cualquier archivo MAP. Contiene todas las "brushes" que no son entidades especiales. En un nivel típico, worldspawn tendrá paredes, suelos, techos, etc. No importa el tamaño del nivel, debe tener un worldspawn. Esta entidad no sólo contiene la geometría del nivel, sino también la información de versión del archivo MAP y los archivos WAD usados. La versión de archivo MAP se da en la propiedad **mapversion**. Los archivos WAD se almacenan en la propiedad **wad**. Los diferentes archivos WAD se separan mediante el uso de punto y coma (;).

Las “brushes” tienen cuatro o más caras. No hay límite acerca del número de “brushes” que puede haber en una entidad.
Estas “brushes” siempre son convexas, lo cual es una propiedad muy útil a la hora de efectuar operaciones geométricas. En vez de describirlas como un conjunto de polígonos, las “brushes” se describen como un conjunto de caras. Un ejemplo de cara es:

```
( 0 64 64 ) ( 64 64 64 ) ( 64 0 64 ) BCRATE02 [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
```

Comienza con **tres puntos que describen un plano**. A continuación tenemos el **nombre de la textura**. Luego se dan los ejes U y V. Después, la rotación de la textura, que es información inútil porque los ejes de textura ya están rotados. Por último, se da la escala de textura U y V.

Aclaremos lo que son los ejes de textura. El primero es el eje U y el segundo es el eje V. Los tres primeros números de un eje de textura dan la normal del eje de textura. El cuarto número da el desplazamiento de la textura. Cada eje de textura puede imaginarse como un plano.

Un archivo MAP típico consistirá en un gran worldspawn y muchas entidades representando luces, puertas, interruptores, disparadores, etc. Un ejemplo final de archivo MAP muestra múltiples entidades y “brushes”. Contiene una habitación, una luz, un objetivo, y una posición de inicio de jugador. Lo importante no es qué hace o qué significa cada entidad, sino la sintaxis del archivo.

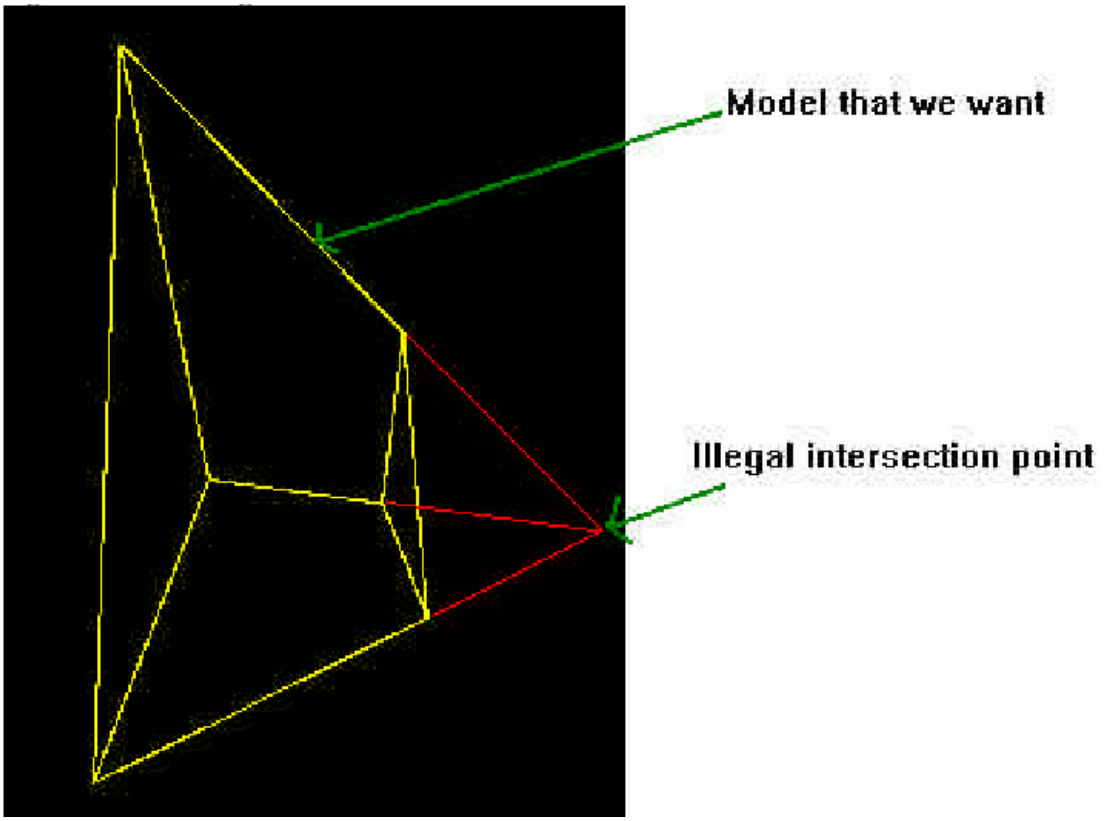
```
{
"classname" "worldspawn"
"MaxRange" "4096"
"mapversion" "220"
"wad" "\games\half-life\cstrike\cstrike.wad;\games\half-life\valve\halflife.wad"
{
( 0 0 256 ) ( 0 256 256 ) ( 256 256 256 ) AAATRIGGER [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 0 256 224 ) ( 0 256 256 ) ( 0 0 256 ) AAATRIGGER [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 256 0 224 ) ( 256 0 256 ) ( 256 256 256 ) AAATRIGGER [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 256 256 224 ) ( 256 256 256 ) ( 0 256 256 ) AAATRIGGER [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 0 0 224 ) ( 0 0 256 ) ( 256 0 256 ) AAATRIGGER [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 0 256 224 ) ( 0 0 224 ) ( 256 0 224 ) C1A0_W2 [ 1 0 0 0 ] [ 0 -1 0 160 ] 0 2 1.6
}
{
( 0 256 0 ) ( 0 0 0 ) ( 256 0 0 ) AAATRIGGER [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 0 0 32 ) ( 0 0 0 ) ( 0 256 0 ) AAATRIGGER [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 256 256 32 ) ( 256 256 0 ) ( 256 0 0 ) AAATRIGGER [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 0 256 32 ) ( 0 256 0 ) ( 256 256 0 ) AAATRIGGER [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 256 0 32 ) ( 256 0 0 ) ( 0 0 0 ) AAATRIGGER [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 0 0 32 ) ( 0 256 32 ) ( 256 256 32 ) C1A0_LABFLRB [ 1 0 0 0 ] [ 0 -1 0 128 ] 0 2 2
}
{
( 0 0 224 ) ( 0 0 32 ) ( 0 256 32 ) AAATRIGGER [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 0 256 224 ) ( 0 256 32 ) ( 32 256 32 ) AAATRIGGER [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 32 0 32 ) ( 0 0 32 ) ( 0 0 224 ) AAATRIGGER [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 0 0 224 ) ( 0 256 224 ) ( 32 256 224 ) AAATRIGGER [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 0 256 32 ) ( 0 0 32 ) ( 32 0 32 ) AAATRIGGER [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 32 256 224 ) ( 32 256 32 ) ( 32 0 32 ) C1A0_W1D5 [ 0 1 0 0 ] [ 0 0 -1 26.6667 ] 0 2
1.2
}
{
( 256 256 224 ) ( 256 256 32 ) ( 256 0 32 ) AAATRIGGER [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 224 256 32 ) ( 256 256 32 ) ( 256 256 224 ) AAATRIGGER [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 256 0 224 ) ( 256 0 32 ) ( 224 0 32 ) AAATRIGGER [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 224 256 224 ) ( 256 256 224 ) ( 256 0 224 ) AAATRIGGER [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 224 0 32 ) ( 256 0 32 ) ( 256 256 32 ) AAATRIGGER [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 224 0 224 ) ( 224 0 32 ) ( 224 256 32 ) C1A0_W1D5 [ 0 1 0 0 ] [ 0 0 -1 26.6667 ] 0 2
1.2
```

```
}
{
( 32 256 32 ) ( 224 256 32 ) ( 224 256 224 ) AAATRIGGER [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 32 224 224 ) ( 32 256 224 ) ( 224 256 224 ) AAATRIGGER [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 224 256 32 ) ( 32 256 32 ) ( 32 224 32 ) AAATRIGGER [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 32 224 32 ) ( 32 256 32 ) ( 32 256 224 ) BCRATE02 [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 224 256 224 ) ( 224 256 32 ) ( 224 224 32 ) BCRATE02 [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 224 224 32 ) ( 32 224 32 ) ( 32 224 224 ) C1A0_W1D5 [ 1 0 0 -21.3333 ] [ 0 0 -1
26.6667 ] 0 1.5 1.2
}
{
( 224 0 32 ) ( 32 0 32 ) ( 32 0 224 ) AAATRIGGER [ 1 0 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 224 0 224 ) ( 32 0 224 ) ( 32 32 224 ) AAATRIGGER [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 32 32 32 ) ( 32 0 32 ) ( 224 0 32 ) AAATRIGGER [ 1 0 0 0 ] [ 0 -1 0 0 ] 0 1 1
( 32 0 224 ) ( 32 0 32 ) ( 32 32 32 ) BCRATE02 [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 224 32 32 ) ( 224 0 32 ) ( 224 0 224 ) BCRATE02 [ 0 1 0 0 ] [ 0 0 -1 0 ] 0 1 1
( 32 32 32 ) ( 224 32 32 ) ( 224 32 224 ) C1A0_W1D5 [ 1 0 0 -21.3333 ] [ 0 0 -1 26.6667
] 0 1.5 1.2
}
}
{
"classname" "info_player_start"
"angles" "0 0 0"
"origin" "64 64 80"
}
{
"classname" "light"
"_light" "255 255 128 200"
"target" "Light01.Target"
"targetname" "Light01"
"origin" "128 128 208"
}
{
"classname" "info_target"
"targetname" "Light01.Target"
"origin" "128 128 32"
}
```

Los archivos MAP se convierten en listas de entidades. Cada entidad debería tener propiedades y polígonos. Nótese que esto no difiere mucho de lo que son en realidad los archivos MAP. El problema es que en vez de almacenar la geometría como una lista de polígonos, los archivos .MAP consisten en “brushes”, que son poliedros convexos definidos por planos. Así que la tarea de un parser de archivos MAP es cargar el archivo MAP en memoria, convertir las “brushes” de una entidad en un conjunto de polígonos y añadirlo a la lista de entidades transformadas.

Una cuestión que surge ahora, es cómo de complicados pueden ser los objetos y edificios construidos si los poliedros son convexos. Primero, cada entidad puede tener más de uno. Si colocamos dos poliedros convexos uno junto al otro, podemos formar poliedros cóncavos más complicados. Sin embargo, habrá polígonos dentro del nuevo objeto.

Imaginemos que colocamos dos cubos, de forma que se están tocando. Forman un poliedro rectangular que más largo en uno de sus ejes. Sin embargo, los lados que están juntos están dentro del nuevo objeto. En primer lugar, es un gasto de memoria guardar estos polígonos, porque nunca se verán. En segundo lugar, este tipo de geometría puede ser ilegal para la compilación de BSP’s.



Este problema se puede solucionar utilizando la operación de unión de CSG. La operación de unión eliminará cualquier polígono que haya dentro de un objeto y nos permitirá unir dos poliedros mientras borra la geometría innecesaria.

Otro problema que tendremos que hacer frente será la aproximación. El uso del tipo 'double' en vez de 'float' nos dará mayor aproximación. Aun así, los errores de redondeo avanzan y, eventualmente las coordenadas pueden estar poco aproximadas. Para luchar contra este inconveniente, se introducen épsilons. En vez de hacer comprobaciones con valores precisos como el cero, comprobamos si el valor está dentro de un cierto rango cercano al cero.

Debido a la complejidad de esta etapa, en vez de programar nuestro propio parser, hemos utilizado el que viene con el editor de mapas de Half-Life, Valve Hammer.

3.4.3 Archivos WAD

Características

Los archivos en formato WAD3 son los usados por Half-Life para almacenar en un único archivo múltiples texturas.
Los archivos WAD inicialmente surgieron en el Doom, y en dicho archivo también se almacenaba, aparte de las texturas, todo el mapa del juego. El WAD3 permite también almacenar otros archivos, tales como archivos de sonidos para el nivel u otras informaciones adicionales (parte que no trataremos aquí)
Las texturas almacenadas en el WAD3 son imágenes de 8 bits con dimensiones que deben ser múltiplos de 16, el tamaño mas pequeño es 16x16 y el mayor 256x256. Cada textura tiene a su vez su propia textura de 24 bits y cuatro niveles MIP.
Los nombres de las texturas pueden ser de 15 caracteres, no pudiendo contener espacios en blanco. Las texturas, al menos para los WAD del Half-Life tienen los nombres en mayúsculas, por lo que al cargar un mapa por ejemplo, antes de realizar la búsqueda se debe convertir el nombre de la textura que se quiere buscar. Por otra parte hemos encontrado texturas y niveles que provienen de juegos basados en el Half-Life, que no siguen esta norma, por lo que la mejor solución, que es la que hemos usado, es la búsqueda en el archivo WAD con los dos métodos, por el nombre sin pasar a mayúsculas y también con el nombre en mayúsculas.
Las texturas que sean animaciones, transparencias, etc... se realizan en ejecución, no teniendo efecto en el WAD. Las transparencias se realizan especificando un color como el color transparente y las animaciones se almacenan separando sus frames, por ejemplo si la animada textura se llama *nombre*, existiran las texturas *A0_nombre*, *A1_nombre*, *A2_nombre*, *A3_nombre*…

Formato del archivo

El formato del archivo WAD3 se divide en 3 partes: la cabecera WAD3, los objetos almacenados y la lump table.
Para poder acceder a cada parte del archivo, se almacenan los offsets correspondientes, ya que hay partes que tienen tamaños variables. Los offsets se refieren al comienzo del archivo.

```
//Cabecera WAD3
struct wadinfo_t
{
        char        identification[4];      //Debe ser WAD2 o WAD3
        int         numlumps;               //Número de Lumps
        int         infotableofs;           //Offset a la Lump table
};

//Entrada de la Lump Table
struct lumpinfo_t
{
        int         filepos;                //Offset a la posición del objeto
        int         disksize;               //Tamaño del objeto en disco
        int         size;                   //Tamaño del objeto descomprimido
```

```
	char		type;			//Tipo de objeto
	char		compression;		//Tipo de compresión
	char		pad1, pad2;		//Relleno
	char		name[16];		//Nombre
};

#define		MIPLEVELS  4 //Niveles de MIP, por defecto 4, pero se podría cambiar

//Objeto textura MIP
struct miptex_t
{
	char		name[16];		//Nombre
	unsigned	width, height;		//Dimensiones de la textura
	unsigned	offsets[MIPLEVELS];
			//Array con los offsets a la información de las texturas

};
```

La cabecera WAD3 indica el tipo de archivo que tenemos (util si aparece por ejemplo un WAD4) y la cantidad de lumps y el offset a la posición donde se encuentra la tabla.
La Lump Table almacena la información de los objetos almacenados en el WAD, incluyendo el offset a la posición donde se encuentra la información del objeto. Dichos objetos pueden ser de otro tipo aparte de texturas, como se ha indicado antes.
Las texturas tienen 2 partes diferenciadas, una que es de tamaño fijo, que contiene la información basica de la textura, altura, anchura, nombre y offsets. La segunda parte es donde se almacena la textura propiamente dicha y tiene un tamaño variable en función de la resolución que tenga esta.

Los niveles MIP son la representación de la textura, en diferentes resoluciones, por si se quiere poder configurar la calidad de la imagen y así usar las texturas de menor resolución para consumir menos recursos en ordenadores menos potentes, o simplemente para poner texturas de menor resolución a objetos que estén siempre muy alejados de la cámara.
La textura tiene 4 MIP si la resolución de esta es por ejemplo de 256x256, el MIP de nivel 0 tiene la resolución propia de la textura y los otros 3, se van dividiendo por 2. Asi la textura de 256x256 tiene los MIPs de tamaño 256x256, 128x128, 64x64 y 32x32 (dividir por 2^nivel_textura, siendo el primer nivel 0).

Así, el objeto textura tiene la siguiente forma:

- Nombre de la textura (16 char)
- Altura (unsigned)
- Anchura (unsigned)
- Array de offsets a los niveles de datos (4 unsigned) [offsets respecto al inicio del objeto!!]
- Datos nivel 1 (alto*ancho bytes)
- Datos nivel 2 (alto*ancho/4 bytes)
- Datos nivel 3 (alto*ancho/16 bytes)
- Datos nivel 4 (alto*ancho/64 bytes)
- Tamaño de la paleta (short)
- Paleta de colores (3 bytes por cada color almacenado)

El tamaño de la paleta de colores debe ser siempre 256
Para acceder a la paleta de colores, hay que sumar los tamaños de la cabecera del objeto los tamaños de los MIP que se encuentran antes de la paleta y saltarse el campo donde se especifica el tamaño. Una forma sería sumar: offset al objeto + offset al último MIP almacenado + tamaño último MIP + tamaño del campo donde se especifica el número de colores de la paleta.

Acceso a la información de una textura

Para extraer la información de una textura en formato RGB para luego poder usarla en el motor gráfico sería el siguiente:

1. Recorrer la lump table hasta encontrar el objeto deseado.
2. Acceder con el offset obtenido a la posición del archivo donde está el objeto textura
3. En el objeto, obtener los datos del nivel de resolución deseado (lo mas normal es acceder al primero).
4. Accederemos al píxel deseado y en la información almacenada, obtendremos un número de 8 bits que se refiere al número de color a usar de los que están en la paleta de colores de la textura.
5. Por último, tendremos que acceder en la paleta la información del color en RGB: palette[colornum*3+rgb] siendo palette un puntero al comienzo de la paleta de colores y rgb 0 para obtener el rojo, 1 para el verde y 2 para el azul. Los valores de cada componente rgb están entre 0 y 255

Búsqueda de una textura

Para buscar una textura una vez obtenido el nombre, hay que obtener el indice de dicha textura dentro del WAD. Esto se hace en la función int WadFile::getTextureIndex(char* name)

Para que la búsqueda sea mucho mas eficiente, ya que el nombre de la textura tiene solo 16 caracteres, se divide en 4 partes y se almacenan en variables de tipo integer:

```
v1 = *(int *)cleanname;
v2 = *(int *)&cleanname[4];
v3 = *(int *)&cleanname[8];
v4 = *(int *)&cleanname[12];
```

Al acceder al nombre almacenado en el archivo, también realizaremos la misma operación, de esta manera se reducen las comparaciones que hay que realizar a 4. (Si usásemos el strcmp se harían 16)

3.4.4 Archivos BSP

Los archivos BSP son generados a partir de los Maps, tras parsearlos y compilarlos con las utilidades qcsg y qbsp2 de valve.
Tras las conversiones, en el archivo se pueden cargar las entidades especificadas en el MAP y luego de los modelos se puede acceder a toda la información respecto a sus caras (aristas que las componen, texturas y coordenadas de textura, etc)
Para un uso mas cómodo de dicho Archivo, hemos implementado unas clases de C++ para acceder a su información basándonos en el código de las herramientas del Half-Life

3.4.5 Carga de los mapas diseñados con WorldCraft

La carga de los mapas diseñados tiene dos partes diferenciadas, la carga de las entidades y la carga de los modelos gráficos.

Carga de entidades

Las entidades que nos podemos encontrar en un mapa creado para nuestra aplicación son principalmente:

- Worldspawn, que almacena la información general del mapa y el modelo estático del nivel.
- Punto de inicio de los actores en el mapa (javy y el estudiante)
- Posición de donde se colocarán los edificios cuando estos sean necesarios
- Puntos que se usarán para crear el grafo para la búsqueda de caminos en el escenario.
- Posición de las cámaras y dirección a la que deben apuntar en las diversas habitaciones
- Iluminación del mapa y puntos de luz.
- Puntos donde hay que cargar modelos mdl del half-life.
- Otras modelos gráficos que no sean estáticos

La carga de las entidades se realiza después de cargar el mapa en el objeto BSPFile. Para ello, vamos obteniendo cada entidad almacenada mediante un bucle, obtenemos el nombre de la clase la que pertenece, que está almacenada en la propiedad classname y mediante una función a la que pasamos este nombre, obtenemos el puntero a la función que tratará la entidad.
Las funciones que tratan las entidades tienen como parámetro de entrada un objeto en el que se almacena toda la información necesaria para poder procesar la entidad (en principio un puntero a la propia entidad, punteros a los actores, a la escena para poder añadir los gráficos y otros más). La función devuelve un objeto con la información de éxito o de fallo para poder tratarlos correctamente fuera de la función.
Todas las entidades tienen como mínimo la propiedad origin, que indica la posición de esta en el mundo. Otras propiedades muy comunes son el ángulo (usado por ejemplo para especificar la orientación inicial de los actores) o el nombre del fichero a cargar, en otros casos.

En general el tratamiento que hay que realizar a la información obtenida de las entidades no suele ser muy complicado, ya que suelen servir para inicializar estructuras

de control de la aplicación, tales como la búsqueda de caminos, las posiciones de las cámaras y etc.
En el caso de que la entidad tenga un modelo asociado, habrá que usar las funciones correspondientes para convertir dicho modelo al formato del motor gráfico.

```
int entity_t::getModelNum()
{
     char* num=getValueForKey("model");
     if ((num==NULL)||(num[0]!='*')) return -1;
     return atoi(&(num[1]));
}
```

Carga de los modelos gráficos

En el caso de que una entidad tenga modelo gráfico, es necesario convertirlo al formato del motor gráfico.
El principal problema al cargar el mapa fueron los objetos Trimesh que son los que había que usar para guardar los datos. Estos objetos, sirven para crear mallas 3D para los modelos pero con las limitaciones de que solo pueden almacenar triángulos, en un Trimesh solo puede haber una textura y no se pueden añadir o eliminar triángulos del objeto una vez construido.
A su vez, los modelos del mapa están compuestos por caras, que son polígonos convexos de un numero arbitrario de lados (normalmente como mucho 4 ó 5), como cada cara puede tener una textura distinta, la conversión se complica en parte.
Primeramente, los polígonos tenían que ser partidos en triángulos, como estos polígonos van a ser siempre convexos, esta triangulación no es muy difícil de realizar, se limita al siguiente sistema:

- El primer vértice del polígono, se asigna como el primer vértice de todos los triángulos que se generen.
- El resto de los vértices los iremos tomando de 2 en 2, formando así los triángulos que buscábamos.

Así, al dividir un pentágono que tiene los vértices (0,1,2,3,4) , crearíamos los triángulos (0,1,2), (0,2,3),(0,3,4).
Un polígono de N vértices se dividirá siempre en N-2 triángulos.

1. Primer método de carga de los modelos

   El primer sistema que implementamos para realizar la conversión fue sencillamente crear un objeto Trimesh por cada cara del modelo. A dicho trimesh se le pasarían todos los triángulos obtenidos y la textura con sus coordenadas de textura correspondientes a cada vértice.
   Lo bueno de este sistema es lo fácil de implementar que es, el problema es su gran ineficiencia al pintarse el escenario ya que para un mapa de 10000 caras, generaba 10000 Trimesh

2. Segundo método de carga de los modelos

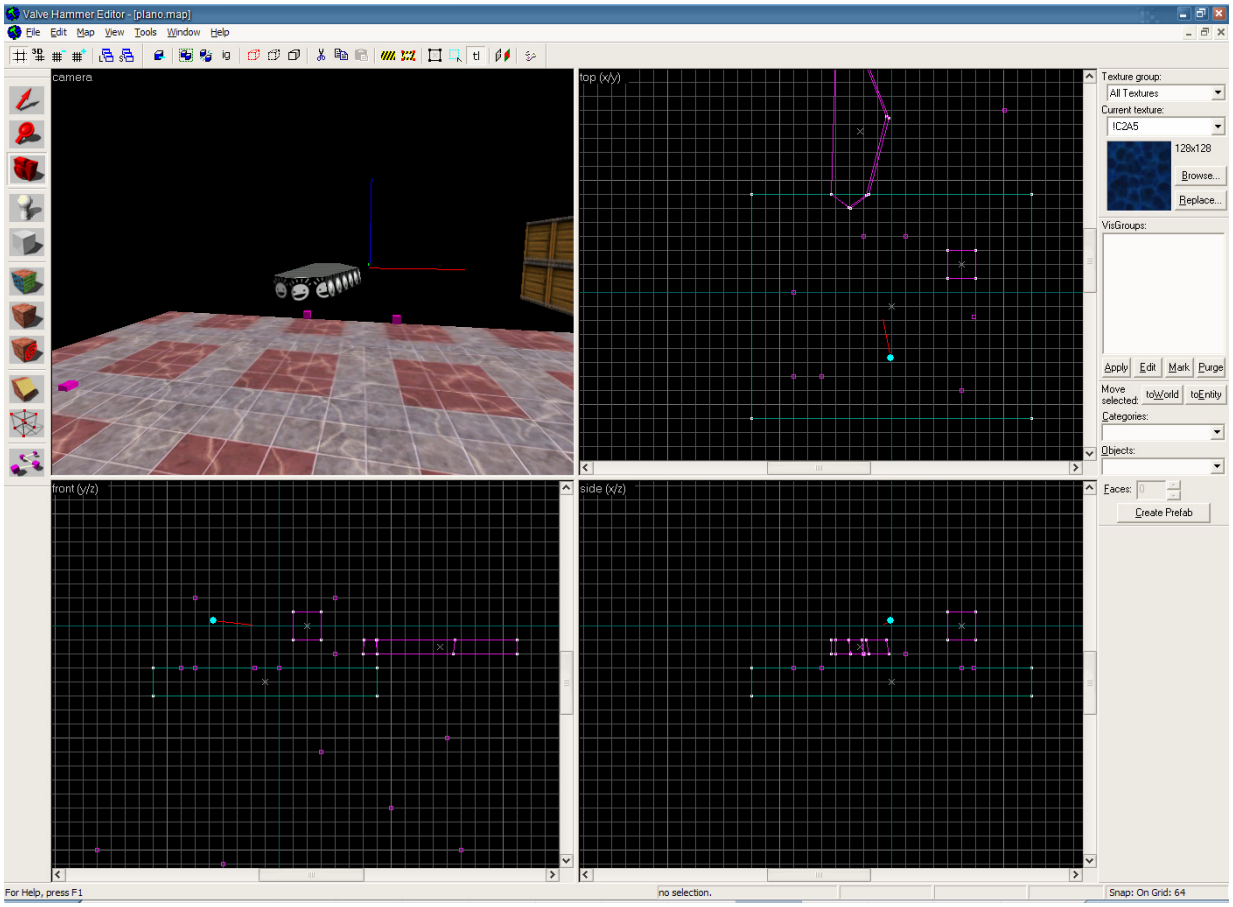
La segunda idea que tuvimos fue hacer lo mismo que la vez anterior, pero en el caso de que se leyesen varias caras seguidas con la misma textura, se juntarían en un solo Trimesh. La ventaja es que había muchas veces que varias caras de un mismo objeto estaban definidas una seguida de otra, con la misma textura y así se ahorraban Trimeshes.
Con este sistema, a un mapa con 10000 caras, se le reducían sus Trimesh a unos 3000, aumentando aceptablemente la velocidad del motor.

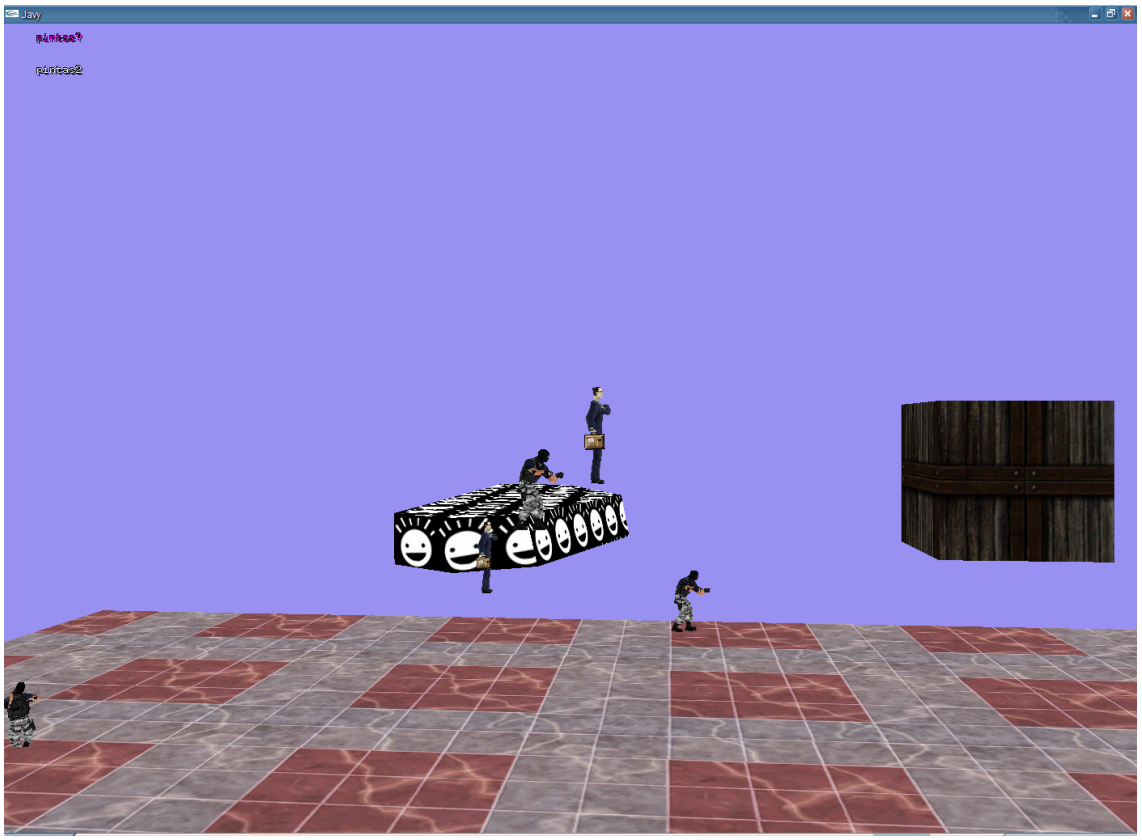
3. Método definitivo para la generación de modelos

Como queríamos mejorar aun mas la velocidad y habíamos leído que las tarjetas gráficas rinden más cuanto mayor numero de triángulos de una misma textura se tengan que dibujar a la vez, en vez de tener que andar cambiando de textura, cambiamos la estrategia radicalmente.
Antes de generar ningún Trimesh, recorremos la lista de caras, creándonos por cada Textura utilizada en el modelo, un vector con las caras que tienen dicha textura. (esto lo hacemos con las clases map y vector de las stl)
Una vez recopilada la información, generamos un solo Trimesh para cada textura, con todas las caras que hemos detectado que usan dicha textura.
De esta última manera, que es la que se puede consultar en el código de la aplicación, se genera un numero de Trimeshes igual al número de texturas, así para el mapa indicado antes, de 10000 caras, solo generaba entre 80 y 90 Trimeshes, que comparado con el primer sistema se ve que es una diferencia abismal.

Carga de las texturas

Los archivos Wad donde se encuentran las texturas utilizadas por los mapas, son especificadas en la propiedad Wad de la entidad Worldspawn. En nuestro caso, para mayor comodidad, únicamente tomamos de dicha propiedad el nombre del archivo, borrando la ruta donde tendría que buscarlo. Así, siempre se busca en el directorio donde está el mapa.
El sistema para cargar la textura es bastante simple, únicamente se limita a intentar cargar de cada Wad la textura deseada, en el caso de que no devuelva ninguna, supone que no la ha encontrado.
Por último reseñar que para evitar cargar y descargar los Wad cada vez que se intenta cargar una textura, hemos implementado una especie de patrón proxy, de forma que se carga solo la primera vez y el resto del tiempo se quedan las clases cargadas en memoria.
Para las texturas en sí este sistema no es necesario, ya que ya lo implementa el propio motor.


Valve Hammer Editor - [plano.map]
File Edit Map View Tools Window Help
camera
top (x/y)
front (y/z)
side (x/z)
Texture group:
All Textures
Current texture:
IC2A5
128x128
Browse...
Replace...
VisGroups:
Apply Edit Mark Purge
Move selected: toWorld toEntity
Categories:
Objects:
Faces:
Create Prefab
For Help, press F1
no selection.
Snap: On Grid: 64


Jawy
player1
player2

3.5 Colisiones y simulación física

La física proporciona una gran cantidad de leyes que pueden ser utilizadas para conocer y simular el comportamiento de los objetos en el mundo real. Las leyes de Newton pueden utilizarse para conocer la reacción de los cuerpos ante la aplicación de fuerzas, la cinemática nos permite conocer sus posiciones en función de sus velocidades, etc.

A partir de todas esas leyes, construir sistemas de realidad virtual en los que se simule el mundo real con exactitud no debería ser muy complicado. Sin embargo, no hay que olvidar que deseamos que el sistema responda en tiempo real. Simular con exactitud cualquier proceso (por ejemplo el movimiento de las olas en una superficie líquida) suele requerir cálculos complejos que requieren un tiempo de cálculo excesivo. Por lo tanto, suelen realizarse grandes simplificaciones que limitan la aplicabilidad de las aproximaciones, pero que permiten realizar dichos efectos en tiempo real. El sistema deja, en muchas ocasiones, de realizar una simulación física, para limitarse a tratar de *imitar* cierto suceso que ocurre en el mundo real.

Incluso al simular hechos sencillos aparecen complicaciones añadidas. Pongamos como ejemplo la caída de objetos simples como esferas. La forma más sencilla de simularlo es controlar en todo momento la posición y la velocidad de la esfera, de modo que antes de comenzar a dibujar un fotograma se pida al simulador físico del sistema que calcule la nueva posición y velocidad en función del tiempo transcurrido desde la última vez. Si la esfera llega al suelo (altura 0), el simulador tendrá que invertir la velocidad (para que la esfera comience a ir hacia arriba), aunque disminuyéndola ligeramente para simular la pérdida de energía de la esfera debido al golpe. Para conseguir todo esto, se pueden utilizar las leyes de la cinemática.

Pero aquí aparece un problema debido a que la simulación no es continua. El sistema podría perder el paso de la esfera por la altura 0, pasando de golpe a tener un valor negativo. Evidentemente esta situación es fácil de controlar (basta modificar la condición de comprobación ligeramente). Sin embargo el cálculo de la posición y velocidad de la esfera se complica considerablemente. Para conseguir el mayor realismo posible, el sistema tendría que resolver una ecuación de segundo grado para obtener el instante de la colisión, calcular la velocidad en dicho instante, modificarla (invertirla y cambiar su módulo para simular la pérdida de energía), y calcular la nueva posición y velocidad de la esfera en función del tiempo transcurrido desde la colisión.

Naturalmente esta dificultad adicional se puede afrontar, pero demuestra la aparición de complicaciones no previstas al simular sistemas continuos de forma discreta. Los nuevos problemas que se generan pueden llegar a ser bastante complicados de solucionar en sistemas más complejos, y en ocasiones añaden inexactitudes según avanza el tiempo de simulación que podrían ser inadmisibles. La solución pasa, naturalmente, por realizar la simulación de forma continua, pero esto no siempre es posible si se desea una respuesta en tiempo real.

La simulación por ordenador es utilizada en un ancho rango de aplicaciones con distinto grado de dificultad, como juegos, programas de modelado, o los simuladores de enseñanza. Sin embargo, para el proyecto tratado aquí no es necesario ningún tipo de simulación física, por lo que no se profundizará en este tema.

Lo que sí es necesario en prácticamente todos los sistemas de entornos virtuales es la detección de colisiones. Por ejemplo los entornos inmersivos, necesitan detectar cuando la mano virtual del usuario colisiona con los objetos del entorno, y los no inmersitos necesitan saber cuando el usuario colisiona con algún objeto al desplazarse dentro del entorno. La detección de colisiones tiene una sencilla aproximación. Si se desea saber si un objeto colisiona con otro, podría ser suficiente mirar si algún triángulo del primer objeto se corta con alguno de los del segundo. Comprobar si un triángulo en 3D colisiona con otro requiere comprobar si alguna de las aristas de uno de ellos colisiona con el contrario, y viceversa. Naturalmente, eso hace a esta técnica totalmente inapropiada para entornos con un alto número de triángulos, especialmente en sistemas en tiempo real.

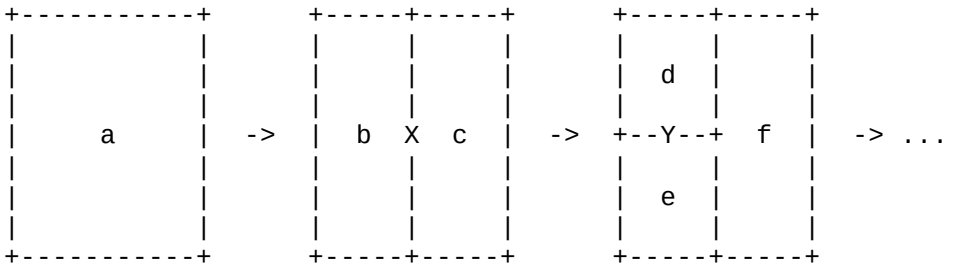
Se han propuesto muchas otras alternativas, que se basan en la división jerárquica de los objetos. Eso permite averiguar muy rápidamente si dos objetos *no* colisionan entre sí; sin embargo, si se detecta colisión es necesario recorrer las estructuras jerárquicas para poder ir refinando progresivamente las comprobaciones. Naturalmente estas técnicas son mucho más rápidas que la anterior basada en *fuerza bruta.* Sin embargo, todas ellas tienen el inconveniente de requerir un preprocesado previo, durante el cual se analizan los objetos y se construyen las estructuras jerárquicas asociadas, que serán utilizadas posteriormente para detectar las colisiones. Aunque esto no debería ser un problema en entornos estáticos, en aquellos en los que los objetos cambian a lo largo del tiempo supone una carga adicional de trabajo al requerir recalcular las estructuras. Ha de quedar claro que, en este caso, cuando decimos entornos dinámicos nos referimos no a entornos donde los objetos se muevan o roten, si no a aquellos donde los objetos se *deformen* de modo que los polígonos y vértices que los constituyen se modifiquen.

3.5.1. Binary Space Partitioning Trees

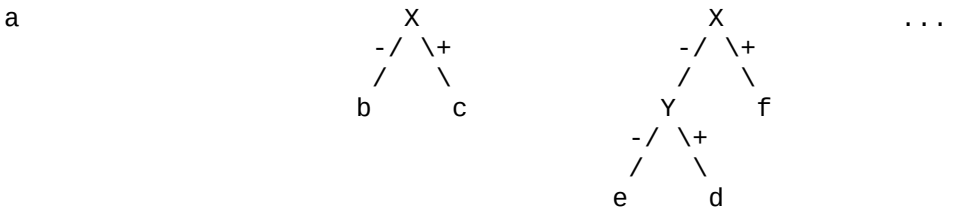
Los árboles BSP (Binary Space Partitioning Trees) representan una estructura recursiva de particionamiento jerárquico, o subdivisión, de un espacio n-dimensional en subespacios convexos. La construcción de árboles BSP es un proceso que toma un subespacio y lo divide mediante un hiperplano que corta el interior de ese subespacio. Así tenemos dos nuevos subespacios que pueden ser divididos nuevamente con una aplicación recursiva del método.

Un hiperplano de un espacio n-dimensional es un objeto n-1 dimensional que puede usarse para dividir el espacio en dos subespacios. En nuestro caso, en un espacio tridimensional, el hiperplano es un plano. Si estuviéramos trabajando con un espacio bidimensional, el hiperplano usado sería una recta.

Una forma sencilla de comprender los árboles BSP es limitarse a estudiarlos en dos dimensiones. Para simplificar la situación, digamos que usaremos sólo rectas paralelas al eje X o al eje Y, y que dividiremos los espacios equitativamente en cada nodo. Por ejemplo, dado un cuadrado en algún lugar del plano XY, elegimos la primera división, y por tanto la raíz del árbol, para cortar el cuadrado por la mitad en la dirección del eje X. En cada parte, elegimos una línea en la dirección opuesta, así que la segunda división partirá cada mitad en la dirección del eje Y. Este proceso sigue recursivamente hasta que llegamos a algún punto de finalización, y tiene este aspecto:



El árbol BSP resultante tiene este aspecto en cada paso:



Cuando se formuló por primera vez el diseño original de los árboles BSP, la idea era utilizarlos para ordenar los polígonos de la escena 3D. La razón para esto es que no existían tarjetas gráficas con Z-Buffer acelerado por hardware, y el uso de esta técnica mediante software era demasiado lento. Hoy en día esta aplicación de los BSP está obsoleta, puesto que la mayoría de las tarjetas incorporan la característica mencionada. En su lugar, se utilizan para otras áreas, como los cálculos de radiosidad, dibujado del mundo y detección de colisiones.

En nuestro caso, vamos a utilizarlas como estructura jerárquica para la detección de colisiones en JAVY, puesto que el motor gráfico utilizado, Wild Magic, incorpora su propio grafo de escena.

3.5.2. Construcción de árboles BSP

Los BSP subdividen el espacio de forma recursiva, tratando cada polígono como un plano divisor a partir del cual clasifican los objetos restantes en tres categorías diferentes: polígonos que pertenecen al plano, polígonos que están delante del plano y polígonos que están detrás. En caso de que el plano elegido corte a alguno de los polígonos, se parte en dos y se inserta cada uno de los polígonos resultantes en el nodo correspondiente. Hay que tener en cuenta que al dividir un polígono puede que uno de los polígonos resultantes tenga un área casi nula; en ese caso, lo descartamos.

Un nodo de nuestro árbol BSP almacena la siguiente información: el plano divisor, los polígonos que pertenecen a dicho plano, un puntero al nodo que representa el subespacio que queda delante del plano y otro que apunta al subespacio que queda detrás.

El algoritmo, en líneas generales, funciona así:

1. Para comenzar la construcción, añadimos a la raíz todos los polígonos de la escena. Una vez hecho esto, comienza el proceso recursivo.
2. De entre la lista de polígonos existentes, seleccionamos uno y tomamos el plano que lo contiene como divisor del espacio. El polígono elegido se inserta en la lista de polígonos coplanarios con el plano divisor.
3. Clasificamos el resto de polígonos con respecto al plano de corte. Es un algoritmo sencillo: clasificamos cada punto del polígono con respecto al plano. Si todos los vértices pertenecen al plano, el polígono es coplanario; si todos están delante, el polígono lo está también; si todos están detrás, el polígono lo está también; en caso contrario, hay que partir el polígono.
4. Cada vez que un polígono es clasificado se añade al nodo hijo correspondiente. El proceso continúa recursivamente en cada nodo nuevo creado.

La selección del plano tiene su importancia. Es deseable tener un árbol balanceado. Sin embargo, esto tiene un coste. Si el plano divisor atraviesa un polígono, hay que partirlo. Una mala selección del plano divisor puede acarrear múltiples divisiones, lo que incrementa el número de polígonos. Normalmente se establece un compromiso entre tener un árbol balanceado y un mínimo número de divisiones.

En cuanto a la finalización del proceso de construcción, hemos optado por terminar cuando todo polígono se ha añadido a algún nodo interno.

3.5.3 Detección de colisiones con árboles BSP

Para saber si un objeto geométrico colisiona con algún polígono de la escena, se debe recorrer el árbol BSP de forma recursiva, comprobando en cada momento en qué lado del plano del nodo actual se encuentra el objeto tratado y descendiendo por el subárbol correspondiente a ese subespacio. Si el plano corta el objeto, hay que calcular la proyección del corte sobre el plano y comprobar si hay colisión con alguno de los polígonos contenidos. Si la hay, la recursión finaliza; si no, se continúa.

3.5.3.1 Colisión de un árbol BSP con un segmento

Si representamos mediante un punto al actor, y buscamos colisiones entre la escena y el segmento que va desde la posición inicial a la posición final del desplazamiento, evitamos el problema de la detección de colisiones en movimiento, puesto que así buscamos colisiones a lo largo de toda la trayectoria del movimiento descrito.

El proceso de detección es muy similar al caso general para detección de colisiones: en cada nodo del árbol se comprueba en que lado está cada extremo del segmento. Si los dos están en el mismo lado, se sigue la búsqueda por ese. En caso contrario, se continúa la búsqueda por el lado en el que está el origen del segmento. Si no se detecta colisión se comprueba con los polígonos inscritos al plano, y si tampoco se detecta esta vez, se continúa con la rama del lado en el que está el final del segmento.

Es importante destacar el orden de la recursión, puesto que así podemos hacer que el algoritmo devuelva el polígono más cercano al origen del movimiento con el que se colisiona.

3.5.3.2 Colisión de un árbol BSP con una esfera

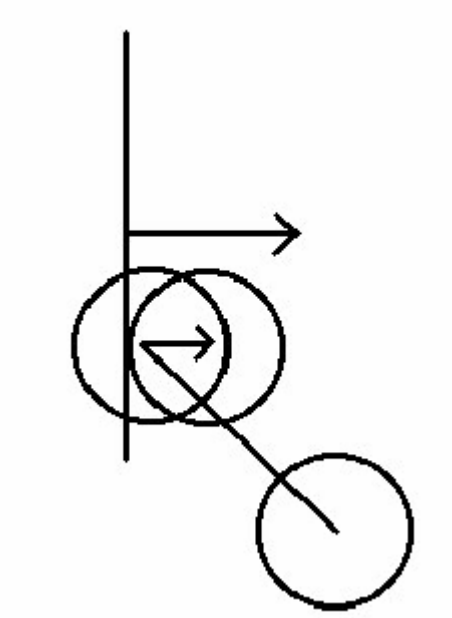
El principal problema del método anterior es que al representar al actor como un punto, perdemos la información sobre su anchura. Debido a esto, el actor podría colisionar con las paredes, ya que su centro no las ha atravesado, pero sí su contorno.

Una posible solución es hacer una aproximación con una esfera, pero correríamos el riesgo de atravesar objetos por culpa de la discretización de las posiciones ante el movimiento

Lo que se ha hecho es tomar una solución mixta: en primer lugar, se ejecuta el test anterior, aproximando el movimiento del actor mediante un segmento, y después se comprueba si la posición final es correcta, aproximando el actor con una esfera. Si en este paso se detecta colisión, se anula el movimiento.

3.5.4 Reacción ante las colisiones

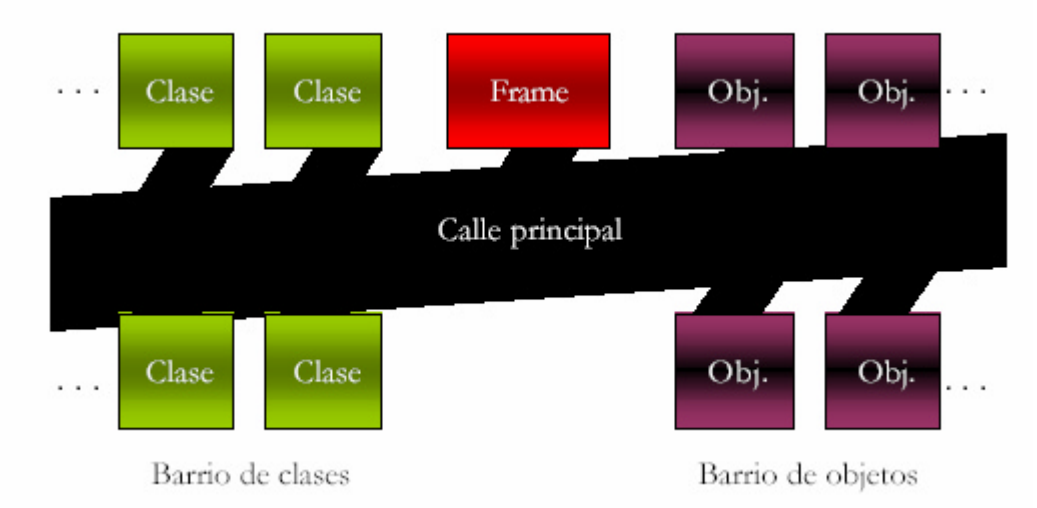
Una posible opción sería que si el personaje choca contra una pared, se detuviera su movimiento. En vez de ello se ha optado por una posibilidad más estética: se "desliza" junto a la pared. Para ello, se calcula el valor absoluto de la distancia de la posición futura, que es incorrecta, con respecto al plano de colisión. A continuación se resta del radio de la esfera del personaje, y esta cantidad se multiplica por la normal del plano. Este vector se sumará a la posición incorrecta si estamos delante del plano; si estamos detrás, se resta.



3.5.5 Implementación

A continuación, vamos a describir cómo es el entorno representado en Javy.

Se trata de un entorno mixto, formado tanto por exteriores como por interiores. El exterior está formado por una única calle larga, con un número variable de edificios a ambos lados. Aunque el número de viviendas cambia dinámicamente, siempre habrá una construcción principal, que se sitúa en lo que se supone el centro de la calle. A esa edificación se le conoce como *frame.* Hacia la izquierda del frame crece el barrio de clases y hacia la derecha el de objetos. El esquema, visto desde arriba, sería algo así:



Los personajes deben desplazarse por el entorno, entrando en el edificio apropiado cuando sea necesario.

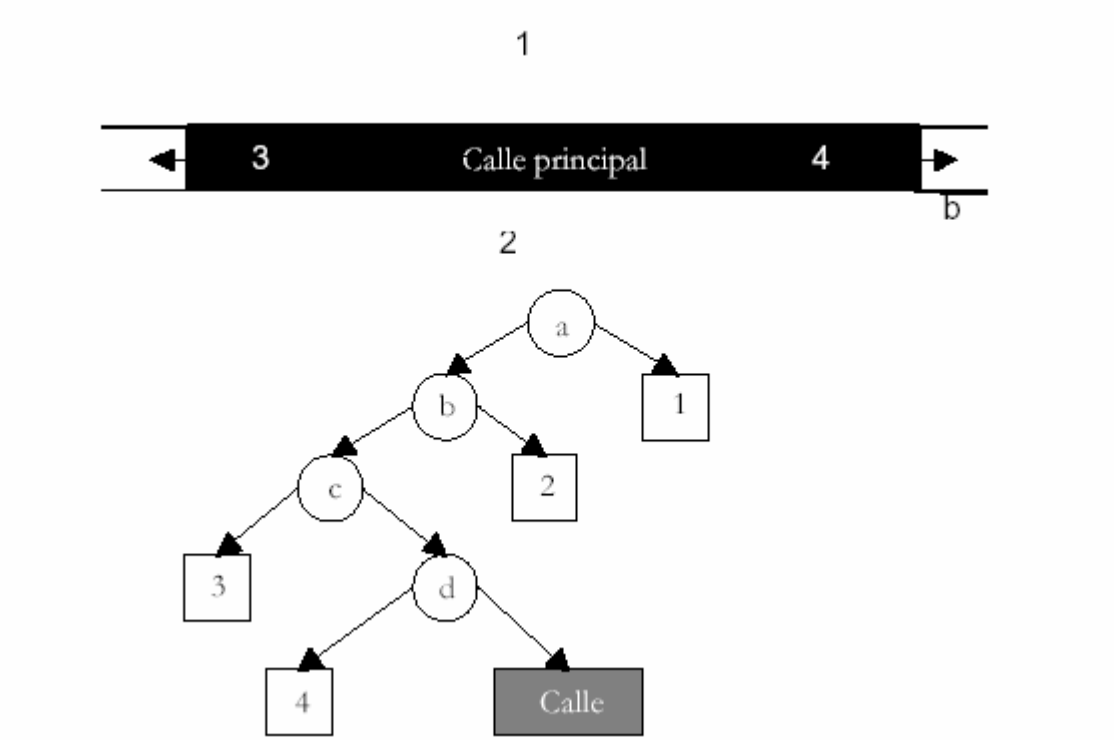
No se han utilizado volúmenes delimitadores para la detección de colisiones, porque deseamos considerar como colisiones algunos desplazamientos en los que el avatar o el agente realmente no chocan con ningún objeto (eso ocurre, por ejemplo, en los extremos de la calle). Al no haber objetos, deberíamos añadir casos especiales al algoritmo de detección de colisiones.

El problema es la estrecha relación existente entre el modelo usado para la representación del mundo y el usado para la detección de colisiones que, en este caso, no resulta apropiado.

La solución es separarlos. Usaremos un árbol BSP con la geometría del mundo para controlar las colisiones.

Antes se comentó que en el proceso de construcción de los BSP, es normal elegir planos de corte que contuvieran algún polígono del mundo. Eso es correcto, pero en nuestro caso, como el mundo tiene una estructura conocida, es posible manipular la construcción del árbol para que se adapte a ella, de modo que resulte más fácil la actualización del BSP en tiempo de ejecución.

Por ejemplo, en la siguiente figura vemos las primeras divisiones hechas, en la que todos los edificios han sido eliminados.



La idea es que de esta forma, cuando vayamos a añadir un edificio, se hará de forma que tengan como predecesores en el árbol a los nodos 1 y 2.

Para poder escoger así los planos, lo que se hace es “marcar” esos planos importantes, como lo son a y b, con el editor de mapas utilizado, Hammer, de Valve LLC. Dicho editor nos permite añadir entidades especiales. De esta forma, al crear el árbol dividiremos la escena con los planos que delimitan la calle, los cuales no se dibujan, pero sí que se usan para las colisiones.

3.6 Control de la cámara:

Actualmente el control de la cámara ha de realizarla el usuario de forma manual. Obviamente esto es muy incómodo, pues no sólo hay que preocuparse de desplazarse por el entorno, si no que además es necesario mover la cámara adecuadamente para no perder de vista al avatar. Es necesario añadir al sistema la capacidad de controlar de forma automática la posición del punto de vista.

Se ha decidido que la cámara sea estática en el interior de los edificios. Es decir, cuando el sistema deba mostrar la parte de dentro de una de las construcciones, la cámara se colocará en un punto estratégico desde el que sean visibles todos los objetos importantes.

Por su parte, en el exterior, la cámara deberá desplazarse en función de la posición actual del usuario. Como el entorno sólo cuenta con una calle, la cámara se moverá a través de ésta siguiendo al usuario. Según se mueva el usuario, la cámara se desplazará siguiéndole dentro de unos límites o cubo imaginario.

Como ya se ha dicho, establecer la cámara supone colocarla en una posición, y establecer la dirección hacia la que "mira". El camino se utiliza para posicionarla, y el punto de vista siempre se dirigirá hacia el avatar. La cámara no podrá ni acercarse ni alejarse en exceso a dicho avatar.

El gestor de la cámara debe ser informado cuando se produce un cambio de zona, es decir cuando el avatar entra o sale de un edificio, para modificar el modo en el que se planea el punto de vista. Para obtener esa información, se averiguará la zona en la que se encuentra el avatar, y el módulo colocará la cámara en función a eso.

El módulo que controlará la cámara se tendrá que ejecutar antes de que el motor gráfico comience a realizar la representación de la escena, para que la posición de la cámara esté siempre actualizada.

Bibliografía

Assarsson, U., Möller, T.: “Optimized View Frustum Culling Algorithms for Bounding Boxes”. *In Journals of graphics tools,5(1), pages 9-22, 2000.*

Bates, J.: “The Nature of Characters in Interactive Worlds and The Oz Project”. *Virtual Realities: Anthology of Industry andCulture, 1993.*

Biggs D. N., Bair G. L., “Pilot Training Device for Underwater Remotely Operated Vehicles”, *In Proc. Del International Training Equipment Conference, The Hague, Netherlands, April 1999.*

Brodlie K., El-Khalili N., Li Y. “Using Web-Based Computer Graphics to Teach Surgery” *En Proc. GVE’99 Computer Graphics and Visualization Education’99*

Bennett F., “Computers as Tutors: Solving the Crisis in Education”, *Book News, Inc. 1999*

Bares W. H., Lester J. C., “Cinematographic User Models for Automated Realtime Camera Control in Dynamic 3D Environments”, *User Modelint ’97, Sardinia, Italy*

Burger, B.; Ondrej, P.; Hasan, M.: “Realtime Visualization Methods in the Demoscene”, *Central European Seminar on Computer Graphics 2002*

Bares W. H., Zettlemoyer L. S., Lester J. C., “Habitable 3D Learning Environments for Situated Learning”, *En Proc. of the Fourth International Conference of Intelligent Tutoring Systems, 1998*

Cunningham S., “GVE ’99: “Report of the 1999 Eurographics/SIGGRAPH Workshop on Graphics and Visualization Education,”

Mark A. DeLoura, “Game Programming Gems”, *Charles River Media, Inc.*

Dede C., Salzman M. C., Loftin R. B., “ScienceSpace: Virtual Realities for Learning Complex and Abstract Scientific Concepts” *En Proc. del Virtual Reality Annual International Symposium 96.*

Duchaineau, M. et al: ROAMing Terrain: Real-Time Optimally Adapting Meshes. Proceedings of the ACM Symposium on Volume Visualization, 1997.

Microsoft DirectX 6.0 Documentation

Eberly, D. H.: “3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics”, *Morgan Kaufmann Publishers , 2000*

Erikson, C., Manocha, D.: “Hierarchical Levels of Detail for Fast Display of Large Static and Dynamic Environments”. *UNC-CH Technical Report TR00-012*

Erikson, C., Manocha, D.: “Simplification Culling of Static and Dynamic Scene Graphs”. *Technical Report TR98-009, University of North Carolina at Chapel Hill, 1998*

Erikson, C., Manocha, D., Baster III, W. V.: “HLODs for Faster Display of Large Static and Dynamic Environments”. *Proc. 2001 Symposium on Interactive 3D Graphics*

Erikson, C. M.: “Hierarchical Levels of Detail to Accelerate the Rendering of Large Static and Dynamic Polygonal Environments”. *Ph.D. dissertation, UNC Chapel Hill, CS*
Department (2000)

Erikson, C.: “Polygonal Simplification: An overview”, *UNC Chapel Hill Computer Science Technical Report TR96-016, 1996*

Foley J. D., van Dam A., Feiner S. K., Hughes J. F., “Computer Graphics, Principles and Practice”, *Addison-Wesley 1990, segunda edición.*

Gamma, E., Helm, R., Johnson, R., Vlissides, J.: “Design Patterns. Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995

“An introduction to Ray Tracing”. *Editado por Andrew S. Glassner. ISBN 0-12-286160-4*

Gottschalk S., Lin M. C., Manocha D., “OBBTree: A Hierarchical Structure for rapid Interference Detection”. *En Proc. de ACM SIGGRAPH 1996*

Gómez M., “Interactive Simulation of Water Surfaces”, *En*

Gobbetti, E., Turner, R.: “Object-Oriented Design of Dynamic Graphics Applications”. *In New Trends in Animation and Visualization, John Willey, 1991.*

Grégoire J. P., Zettlemoyer L. S., Lester J. C.: “Detecting and Correcting Misconceptions with Lifelike Avatars in 3D Learning Environments”, *En Proc. of the Ninth World Conferenceon Artificial Inteligence in Education, pp 586-593, Le Mans, France, Julio 1999*

Hoff III K. E., Culver D., Keyser J., Lin M., Manocha D. “Fast computation of generalized Voronoi diagrams using graphics hardware”, *En Proc. de ACM SIGGRAPH 1999*

Hoff III K. E., Culver D., Keyser J., Lin M., Manocha D. “Interactive Motion Planning Using Hardware-Accelerated Computation of Generalized Voronoi Diagrams”, *En Proc. De ACM SIGGRAPH (1999)*

Kenneth E. Hoff III: “Faster 3D Game Graphics by Not Drawing What Is Not Seen”, *ACM Crossroads, 1997.*

Hubbard P. M., “Approximating Polyhedra with Spheres for Time-Critical Collision Detection” *ACM Transactions on Graphics, 1996*

Loyall, A. B., Bates, J.: “Real-time Control of Animated Broad Agents”. *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society, Boulder, Colorado, June 1993.*

Luebke, D., Georges, C: “Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets”. *ACM Interactive 3D Graphics Conference, Monterey, CA, 1995.*

Leitão J. M., Moreira A., Santos J. A., Sousa A. A., Ferreira F. N., “Evaluation of Driving Education Methods in a Driving Simulator”, *En Proc. GVE'99 Computer Graphics and Visualization Education'99*

Lester J. C., Zettlemoyer L. S., Grégoire J. P., Bares W. H., “Explanatory Lifelike Avatars: Performing User-Centered Tasks in 3D Learning Environments”, *En Proc. of the Third International Conference on Autonomous Agents, Seattle, Washington, 1999.*

Mei â ner, M., Bartz, M., Hüttner, T., Müller, G, Einighammer, J.: “Generation of Subdivision Hierarchies for Efficient Occlusion Culling of Large Polygonal Models”.

Monsieurs P., Coninx K., Flerackers E., “Collision Avoidance and Map Construction Using Synthetic Vision”, *Workshop on Intelligent Virtual Agents (Virtual Agents 99), Salford, United Kingdom*

Melax S. “Dynamic Plane Shifting BSP Traversal”, *En Proc. De Graphics Interface, 2000*

Naylor B. F., “A Tutorial on Binary Space Partitioning Trees”, *En Proc. De Computer Games Developer Conference, 1998*

Michael Putz, Character Animation for Real-time Applications, *Central European Seminar on Computer Graphics 2002*

Rogers, D. F., Adams, J. A.: "Mathematical Elements for Computer Graphics", *2 nd Edition ISBN 0-07-053529-9*

Rabin S. "A* Aesthetic Optimizations", *En* [DeLoura00]

Rabin S. "A* Speed Optimizations", *En* [DeLoura00]

"Robust and Accurate Polygon Interface Detection", en www.cs.unc.edu/~geom/OBB/OBBT.html

Rö ß ling G., Freisleben B., "Approaches for Generating Animations in Lectures", *AACE 11th International Society for Information Technology and Teacher Education (SITE 2000) Conference, San Diego, California.*

Rö ß ling G., Freisleben B., "Experiences in Using Animations in Introductory Computer Science Lectures", *ACM 31 st SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas.*

Rickel, J., Johnson, W. L.: "Task-Oriented Collaboration with Embodied Agents in Virtual Worlds". *In J. Cassell, J. Sullivan, and S. Prevost (Eds.), Embodied Conversational Agents. Boston: MIT Press, 2000.*

Rodger S. H. "Integrating Animation Into Courses". *Conference on Integrating Technology into Computer Science Education, Barcelona, Spain, 1996*

Rö ß ling G., Schüler M., Freisleben B., "The ANIMAL Algorithm Animation Tool", *ACM 5 th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), Helsinki, Finland.*

Segal, M. Akeley, K.: "The OpenGL Graphics System: A Specification (Versión 1.2.1)", 1999

Simpson, Z. B.: "Design Pattern for Computer Games". *Computer Game Developer's Conference, Austin, TX; San Jose C*A.En http://www.totempole.net/patterns/gamepatterns.html

Sýkora, D., Jelínek, J.: "Efficient View Frustum Culling", *Central European Seminar on Computer Graphics, 2002*

Stout B., "The Basics of A* for Path Planning"

Turner R., Balaguer F., Gobbetti E, Thalmann D "Physically-Based Interactive Camera Motion Control Using 3D Input Devices", *En Proc de Computer Graphics International '91*

Turner R., Gobbetti E., Soboroff I., "Head-Tracked Stereo Viewing with Two-handed 3D Interaction for Animated Character Construction", *Eurographics '96*

Watt, A.: “Complex Scene Management”, *en el curso “3D Games Technology, Real Time Rendering and Character Animation” de XV ECI (2001) Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires.*

Welzl, E.: “Smallest enclosing disks (balls and ellipsoids)” *Lecture Notes in Computer Science, New Results and New Trends in Computer Science, 1991*

Watt A., Policarpo F., “3D Games: Real-time Rendering and Software Technology”, *Addison-Wesley ISBN 0201-61921-0*

Zhang, H., Hoff, K. E.: “Fast Backface Culling Using Normal Masks”. *ACM Interactive 3D Graphics Conference, 1997*